uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Electrónica Industrial y Automática
Curso 2017-2018

Trabajo Fin de Grado

"Modelado Virtual y Simulación de Vehículos Autónomos en Gazebo"

Pedro Mascaró Pons

Tutor

Pablo Marín Plaza

Leganés, Junio 2018

RESUMEN

La presente memoria forma parte del proyecto iCab de la Universidad Carlos III de Madrid, que consiste en la en la implementación de vehículos autónomos. El objetivo de este proyecto es modelizar los vehículos y posteriormente realizar simulaciones en entornos virtuales.

Para poder llevar a cabo todos los desarrollos se han utilizado herramientas de código abierto. Se ha utilizado el entorno de ROS para realizar los desarrollos y Gazebo para llevar a cabo las simulaciones.

El desarrollo del proyecto se ha divido en varias etapas; en primer lugar, se ha logrado simular la geometría de Ackermann. A continuación, se ha implementado dicha configuración a un modelo tridimensional que representa el vehículo que se pretende simular. Finalmente, se han añadido todos los sistemas de percepción que están equipados en el coche real.

Una vez modelizado el vehículo, se han realizado diversas simulaciones en Gazebo en un entorno de test que se ha creado. En ellas se puede apreciar que los objetivos que se definen al principio del proyecto se cumplen con éxito.

Palabras clave: vehículo; modelo; ROS; geometría de Ackermann, eslabón; articulación; sistemas de percepción

CONTENIDO

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

El presente trabajo de fin de grado está incluido dentro del proyecto iCab del *Intelligent Systems Laboratory* de la Universidad Carlos III de Madrid, cuyo objetivo es implementar un vehículo que circule de forma autónoma dentro del campus de la propia universidad.

La parte correspondiente de este proyecto es la modelización del vehículo para su posterior simulación en entornos virtuales, la cual se llevará a cabo mediante herramientas de código abiertas y que son gratuitas. ROS, Gazebo y Sketchup.

El objetivo es aportar una herramienta de soporte que permita facilitar y acelerar en la medida de lo posible el desarrollo del proyecto global. Esta herramienta consistirá en una plataforma donde se podrán realizar simulaciones del comportamiento del vehículo. Permitirá evaluar la planificación de trayectorias, así como el funcionamiento de todos los sistemas de percepción.

La repercusión más directa será una reducción en los gastos asociados al proyecto, ya que no se deberá de utilizar el vehículo real para poder llevar a cabo una prueba, con el riesgo de que pueda ocurrir algún desperfecto o que se pueda desgastar algún componente del vehículo. Por este mismo motivo y por la agilidad entre simulaciones, también repercutirá en un ahorro de tiempo.

Una vez finalizado el proyecto, se deberá de realizar una labor de mantenimiento sobre el modelo simulado, añadiendo las funcionalidades que se incorporen al vehículo real, así como actualizando y optimizando el comportamiento de las que estén ya incluidas.

1.1 MOTIVACIÓN

La mayoría de los accidentes de tráfico se pueden imputar a errores humanos debidos por la falta de atención. Esta cantidad probablemente no haga más que incrementarse debido a que los nuevos estilos de vida que constantemente ofrecen toda una serie de posibles distracciones, desde los sistemas multimedia del coche hasta el teléfono móvil. Según varios estudios, como un hecho por L. C. Davis en 2004 [1], “los coches sin conductor reducirán de manera sustancia los accidentes tráfico y atascos; incluso si están circulando entre coches convencionales”. De este modo, los vehículos autónomos, se pueden considerar como una posible solución a esta problemática.

Una vía para poder evaluar y desarrollar las soluciones que pueden ofrecer los vehículos autónomos de manera segura, eficiente y barata es el uso de simuladores. Recientemente, con los avances tecnológicos tan rápidos que han habido en el campo de la informática, los simuladores se han empezado a utilizar para llevar a cabo este tipo de proyectos. Permiten probar más situaciones de las que serían posible en escenarios reales, así como probar situaciones que podrían resultar ser peligrosas para involucrar humanos en ellas. Así pues, analizar sistemas complejos en mundos virtuales es la solución ideal para validar las soluciones de manera rápida, con más posibilidades y mínimo riesgo.

1.2 OBJETIVOS

El objetivo de este proyecto es modelar el vehículo iCab y posteriormente realizar simulaciones en un entorno virtual. Para ello se va a utilizar el software de Gazebo, entorno que permite visualizar los robots definidos por ROS.

El vehículo deberá respetar la geometría de Ackermann a la hora de realizar movimientos de giro para cambiar la dirección. Además, el coche deberá estar equipado con los sistemas de percepción que tiene el vehículo en la realidad, un sensor Velodyne, un sensor laser y una cámara estéreo. Finalmente se va a definir un entorno de pruebas donde se realicen las distintas simulaciones.

Para ello, se han definido una serie de objetivos que se pretenden cumplir una vez realizado el proyecto:

- Simulación de la geometría de Ackermann.
- Creación del modelo en 3D.
- Ajuste del modelo 3D a la geometría de Ackermann.
- Adición del sensor Velodyne.
- Adición de la cámara estéreo.
- Adición del sensor laser.
- Creación de entorno de pruebas.
- Localización del vehículo.
- Simulación del conjunto.

2 ICAB

iCab, en inglés *Intelligent Campus Automobile*, consiste en un carrito de golf propulsado de manera eléctrica, el modelo E-Z-GO. Se ha modificado para que permitir que pueda navegar de forma autónoma y permita planificar trayectorias.

En el vehículo se han realizado modificaciones mecánicas y eléctricas. De este modo, se ha eliminado el volante y en su lugar se ha instalado el motor encoder para controlar la dirección del carrito de manera eléctrica. Además, el acelerador está desactivado, de este modo, la tracción eléctrica se controla mediante un amplificador gobernado por un microcontrolador PIC, tanto la marcha delantera como la trasera. El rotor y el estator del motor son independientes uno del el otro, de este modo se facilita el control de la entrega de potencia y par en distintos medios, como pueden ser pendientes o terrenos rugosos.

Ilustración 1 iCab 1

En cuanto a sistemas de percepción se refiere, el vehículo está equipado con un sensor laser (SICK LMS 291). El dispositivo tiene más de 180 grados de escaneo, con una resolución angular de 0,25 grados. Está situado en el parachoques del vehículo a 30 cm de altura sobre el suelo. Para evitar la detección de las ruedas delanteras, el rango de detección se ha limitado a 100 grados a 20 Hz.

Además, el vehículo está equipado con una cámara binocular de visión stereo. La cámara tiene una resolución máxima de 1032x736 pixels a 20 frames por segundo. Está montada en la parte delantera del vehículo debajo de la cubierta, a una altura de 160 cm y una orientación de -45 grados.

Por último, se ha colocado un sensor de la casa Velodyne LiDAR con el propósito de facilitar la detección de obstáculos y permitir la navegación de forma autónoma. Este sensor, se ha ubicado sobre la cubierta del carrito de golf una altura de 181 cm.

3 MODELO DE ACKERMANN

El modelo o geometría de Ackermann es una disposición geométrica de los ejes de dirección del vehículo que permite realizar trayectorias circulares sobre un mismo punto. Patentada en 1818 por el alemán Rudolph Ackermann, esta geometría fue diseñada para controlar el sistema de dirección de carruajes.

El propósito de esta disposición es evitar el desplazamiento lateral del vehículo al realizar una trayectoria circular. La solución geométrica a este problema se consigue proyectando todos los ejes de giros de las ruedas converjan en un mismo punto en común. A consecuencia de que las ruedas traseras son fijas en el en sentido que no cambia su dirección, el punto de convergencia deberá estar sobre esta línea del eje trasero.

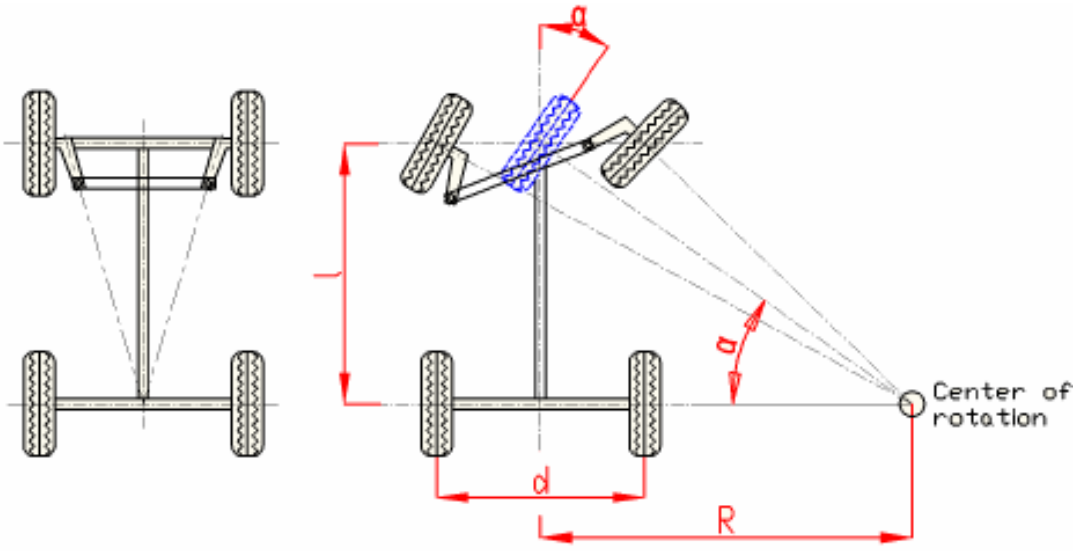


Ilustración 2 Geometría del modelo de Ackermann

De esta manera, se evita que cada rueda tenga su propio pivote de giro y todas tengan el mismo punto común. Esto permite controlar de manera más sencilla la maniobrabilidad del vehículo al enfrentarse a trayectorias complicadas donde se tengan que hacer grandes variaciones en la dirección.

4 ROS

Como su nombre indica, ROS [2] (Robot Operating System) es un meta sistema operativo para el desarrollo y diseño de soporte lógico para robots. ROS provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensaje y administración de paquetes.

Antes de ROS, para el diseño de robots cada desarrollador dedicaba una gran cantidad de tiempo a dedicar físicamente el robot junto a su software asociado, es decir, se requerían habilidades de mecánica, electrónica y programación integrada.

A consecuencia de ello, se creó este entorno que permitiese el intercambio de conocimiento de distintas disciplinas y así permitir un desarrollo más eficiente de los proyectos.

- **Tecnología de par a par**: En inglés, *peer to* peer, ROS se puede ejecutar distintas computadoras, lo que permite crear estructuras complejas.
- **Código abierto**: Al ser de código abierto, se permite el intercambio de información y soporte entre la toda la comunidad, compuesta de investigadores, empresas y aficionados.
- **Flexible**: al tratarse de un sistema código abierto, se trata de una infraestructura adaptable a diversas necesidades.
- **Multilenguaje**: ROS puede ser programada es distintos lenguajes; Python, C++ y Lisp, además de otras librerías experimentales en Java y Lua.

El principio básico de ROS es permitir ejecutar en paralelo una gran cantidad de procesos y ejecutables que deben poder intercambiar información de manera síncrona o asíncrona. Por ejemplo, un sistema debe de estar consultando de manera simultánea la frecuencia definida por sus distintos sensores (distancia por ultrasonidos, temperatura, giroscopio, cámaras…), recopilar la información y actuar conforme los datos de entrada, como por ejemplo por ejemplo podría ser controlar los motores para mover el robot. Todo este proceso se debe hacer de manera continua y en paralelo. Para ello ROS posee su propia estructura [3]:

NODOS: son los procesos que ejecutan la computación, si se quiere que los procesos interactúen unos con otros se necesita crear una red de nodos. Un robot suele estar compuesto por varios nodos. Por ejemplo, un nodo controla el sensor laser, uno controla el motor de las ruedas, uno la localización, otro calcula la trayectoria que se debe seguir, así hasta componer la estructura completa del robot.

MASTER: el master proporciona el nombre y la información necesaria para permitir la conexión con otros nodos. Si no se tiene el sistema, no se puede comunicar con otros nodos, servicios, mensajes y otros. En una arquitectura distribuida, se tendrá el master en un ordenador y se podrán ejecutar los nodos en otras computadoras.

PARAMETER SERVER: permite la posibilidad de guardar los datos de manera centralizada. Con este parámetro, es posible configurar los nodos cuando se están ejecutando.

MESSAGES: los nodos se comunican entre ellos mediante mensajes. Un mensaje contiene datos que entregan información a otros nodos. ROS contiene varios tipos de mensajes, además de que también cada usuario puede desarrollar su propio tipo de mensaje para adaptarlo a sus necesidades.

TOPICS: cada mensaje debe tener un nombre para poder ser direccionado en la red de ROS. Cuando un nodo está enviando información se dice que está publicando un topic. Los nodos puedes recibir topics de otros nodos suscribiéndose a topics. Un nodo se puede subscribir a un topic aunque no haya otros nodos publicando a ese topic, esto permite desvincular la producción de la consumición. Es importante destacar que los nombres de los topics son únicos para poder evitar problemas de confusión entre ellos.

SERVICES: Cuando se publican topics, se está enviando información de diversas maneras, pero cuando se necesita una consulta o una respuesta de un nodo no se puede hacer mediante topics. Los services permiten interactuar entre nodos, cuando un nodo tiene un service, todos los demás nodos se pueden comunicar con él. Del mismo modo que los topics, los services deben tener un nombre único.

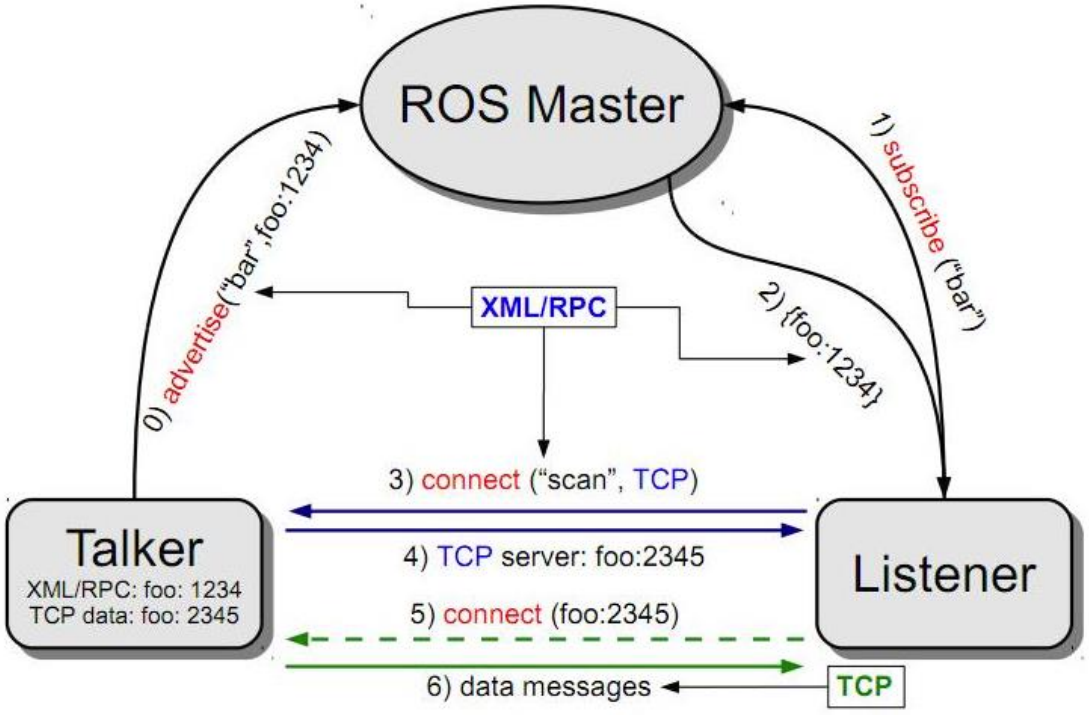


Ilustración 3 Estructura básica de ROS

BAGS: las bags son un formato específico para guardar y ejecutar los mensajes de ROS. Son un importante mecanismo para guardar datos, por ejemplo todos los datos relacionados con un sensor, que puede resultar difícil de recopilar per es necesaria para desarrollar los algoritmos. Las bags se usan en el desarrollo de robots más complejos.

4.1 COMPONENTES ESPECÍFIOS DE ROS

4.1.1 Mensajes estándar de robot:

ROS posee un conjunto de mensajes estándar, *std_msgs*, que permiten cubrir la mayoría de casos de uso planteados en la robótica. Este tipo de mensajes contienen los tipos más primitivos de cualquier lenguaje de programación. Estos formatos permiten definir mensajes más complejos para conceptos geométricos como orientación, transformadas y vectores; respecto a sensores, como cámaras IMUs y láseres; y para navegación, como odometría, trayectoria y mapas; entre otros. Estos han sido desarrollados por la amplia comunidad de ROS, que sigue incluyendo soporte de mantenimiento. Los tipos de mensajes más destacados son los siguientes:

Tipo de mensaje	**Definición**	**Ejemplo**
Bool	bool *data*	*true*
Char	char *data*	P
Empty		
Float32	float32 *data*	23,76
Int32	int32 *data*	12
String	string *data*	asd
Time	time *data*	4’

Tabla 1 Tipos de mensajes principales

Los demás mensajes de pueden ver online en el siguiente enlace:

http://wiki.ros.org/std_msgs

4.1.1.1 ackermann_msgs:

A partir de estos mensajes simples vistos en el apartado anterior se crean tipos de mensajes más complejos, como sería el caso de *ackermann_msgs*, que sirven para poder controlar robots cuyas ruedas delanteras cumplen con la geometría de Ackermann. Su estructura es la siguiente:

Nombre	**Tipo**	**Descripción**
steering_angle	Float32	Ángulo deseado de las ruedas (rad)
steering_angle_velocity	Float32	Variación del ángulo de las ruedas (rad/s)
speed	Float32	Velocidad deseada (m/s)
acceleration	Float32	Aceleración deseada (m/s^2)
jerk	Float32	Variación deseada de la aceleración (m/s^3)

Tabla 2 ackermann_msgs

4.1.2 Librería geométrica de Robot:

Diseñada a fin de ser eficiente, la librería *tf* se usa para coordinar y actualizar todas las transformadas en robots de más de cien grados de libertad. La librería *tf* permite definir ambos tipos de transformadas, estáticas, como a cámara fijada sobre una base fija; y dinámicas, como una junta en un brazo robótico. Se puede transformar los datos de un sensor ente cualquier par de coordenadas del sistema [4].

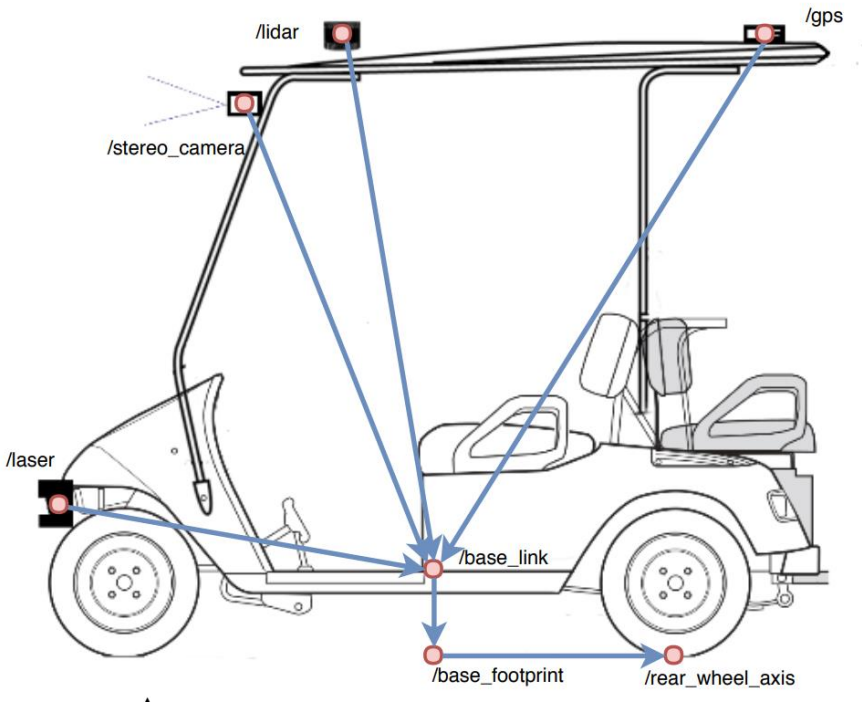


Ilustración 4 Ejemplo de tf

La librería contiene una serie de comandos para visualizar e intercambiar información en ROS:

- **view_frames**: permite visualizar el árbol completo de las transformadas.
- **tf_monitor**: monitoriza las transformadas entre eslabones.
- **tf_echo**: imprime una transformada específica en pantalla.
- **roswtf**: ofrece ayuda a solucionar problemas con tf.
- **static_transform_publisher**: comando para enviar transformadas estáticas.

4.1.3 Lenguaje de descripción del robot:

Otro problema común que ROS proporciona solución es la manera de describir el robot de una manera que pueda ser leída por el ordenador. ROS ofrece un conjunto de herramientas para escribir y modelizar un robot, y de esta manera, puede ser entendido puede ser entendido por el resto del sistema. El formato para describir el

robot en ROS es el URDF (Universal Robot Description Format), que consiste en un documento XML en el que se puede describir las propiedades físicas del robot, desde la longitud de los brazos y tamaño de las ruedas, hasta la localización de los sensores en la parte visual del robot. Una vez definido de esta manera, el robot puede ser fácilmente usado mediante la librería *tf*, renderizada en tres dimensiones para ser visualizado y usado en simuladores.

4.1.4 Diagnósticos:

ROS ofrece una manera estándar de producir, recoger y agregar diagnósticos sobre el robot. De este modo, de un vistazo se puede rápidamente observar el estado del robot y determinar como arreglar las incidencias que surgen en el desarrollo.

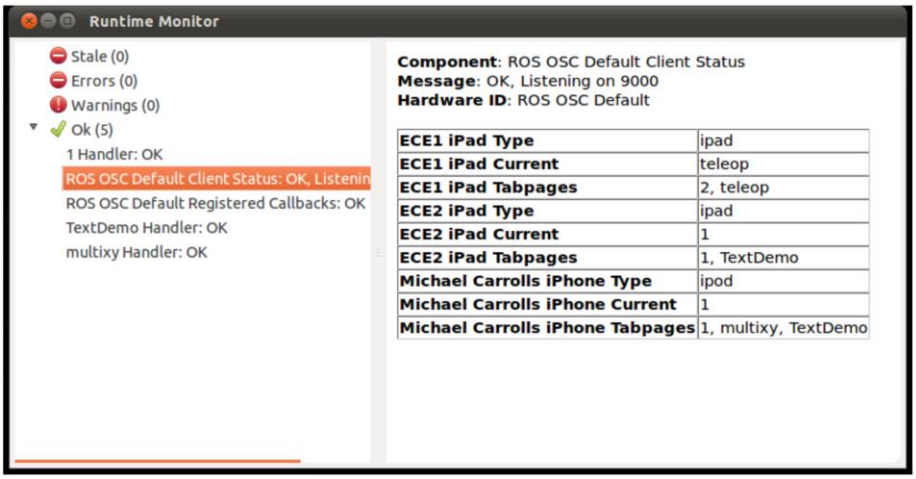


Ilustración 5 Ejemplo de diagnósticos

4.1.5 Paquetes con funcionalidades específicas:

ROS además ofrece algunas "baterías incluidas" o paquetes capaces de ayudar a empezar y enderezar los distintos casos de uso. Existen paquetes de ROS que solucionan problemas básicos de la robótica, como la estimación de la orientación, la localización en el mapa, construir un mapa y hasta navegación móvil. Estas herramientas permiten a los desarrolladores avanzar más rápidamente con menos esfuerzo.

Paquete	**Funcionalidades básicas**
robot_state_publisher	Permite publicar el estado del robot a tf.
twist_mux	Multiplexor de giro, permite priorizar o anular los comandos de velocidad.
controller_manager	Ofrece un bucle en tiempo real que permite controlar los mecanismos del robot.
map_server	Ofrece un mapa como un servicio de ROS.

Tabla 3 Paquetes básicos de ROS

4.2 HERRAMIENTAS DE ROS

Una de las características más fuertes de ROS es el potente conjunto de herramientas de desarrollo. Estas herramientas permiten graficar, visualizar y depurar el estado del sistema cuando se está desarrollando. El mecanismo integrado de publicar/suscribir permite adentrarse en el flujo de información del sistema, facilitando la comprensión de las incidencias que ocurren y su depuración. ROS tiene la ventaja en este asunto a través de una extensa colección de interfaces gráficas y líneas de comando que simplifican el desarrollo y la depuración [4].

4.2.1 Comandos de línea:

ROS puede ser usado al 100% sin interfaz gráfica. Todas las funcionalidades son accesibles a través de alguno de los más de 45 comandos de línea. Estos comandos permiten lanzar grupos de nodos, analizar los topics, services y acciones; grabar y reproducir información; y muchas otras situaciones.

Los comandos de línea que se utilizan de manera más frecuente son los siguientes [5]:

- **roscd**: cambia la ubicación de un paquete de ROS.
- **roscore**: ejecuta todos los elementos que se requieren para dar soporte de ejecución a los sistemas de ROS. Tiene que estar en funcionamiento para posibilitar la comunicación entre nodos.
- **roscreate-pkg**: crea e inicializa un paquete de ROS.
- **rosnode**:
    - *rosnode info*: imprime información del nodo en pantalla.
    - *rosnode kill*: finaliza la ejecución del nodo
    - *rosnode list*: muestra los nodos que se están ejecutando.
    - *rosnode machine*: muestra los nodos activos en la máquina**.**
- **roslaunch**: permite ejecutar cualquier aplicación contenida en un paquete de ROS.
- **rosrun**: permite ejecutar un nodo de ROS.
- **rostopic**:
    - *rostopic bw*: permite mostrar el ancho de banda que consume un topic.
    - *rostopic echo*: imprime los datos del topic.
    - *rostopic find*: encuentra las conexiones de un topic.
    - *rostopic info*: imprime la información de un topic.
    - *rostopic pub*: publica datos a un topic en funcionamiento.
    - *rostopic type*: imprime el tipo de mensajes que debe de contener un topic.
- **roswtf**: permite comprobar si todo está funcionando correctamente

Además, en este enlace se pueden ver y comprobar todos los demás comandos de línea que ofrece ROS:

http://wiki.ros.org/ROS/CommandLineTools

4.2.2 RViz:

Posiblemente es la herramienta más conocida de ros, RViz ofrece una visualización tridimensional de varios tipos de sensores y cualquier robot descrito con una estructura URDF. Además, RViz puede visualizar la mayoría de los mensajes más comunes de ROS, como los escaneos de los láseres, nubes de puntos, e imágenes de cámaras. También usa la información de la librería *tf* para mostrar el conjunto completo en uno entorno tridimensional junto a los a los datos que se recopilan de los sensores. Visualizar todos los datos permite analizar rápidamente todo lo que el robot observa y ayuda a identificar problemas como una mala configuración de los sensores o imprecisiones en el modelo.

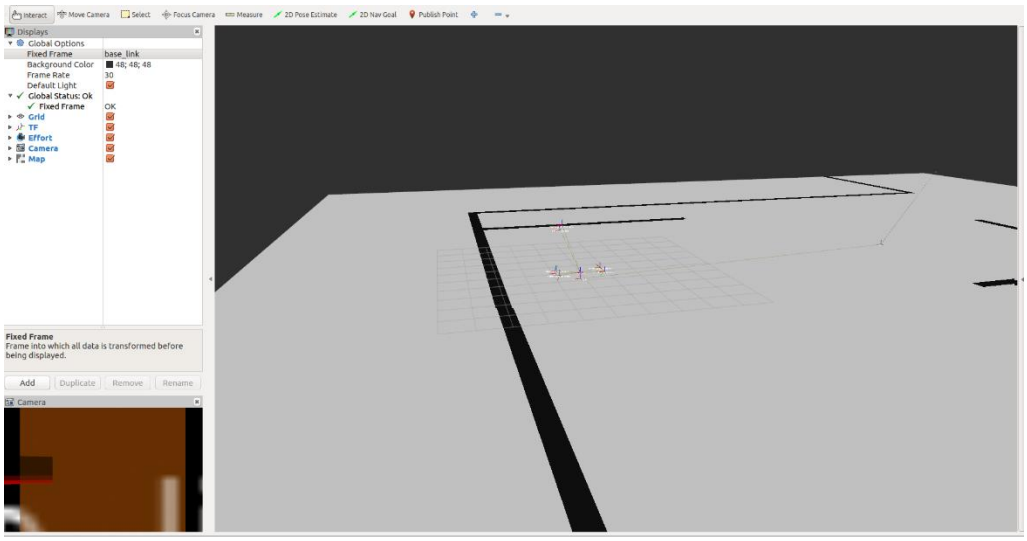


Ilustración 6 Ejemplo de visualización en RViz

4.2.3 RQT:

ROS ofrece RQT, una estructura para visualizar varias características de los robots. Gracias a la posibilidad de añadir varios plugins y la de dividir la pantalla en varias partes (ilustración 7), se pueden visualizar diversos elementos de forma simultánea. Además, cada usuario puede añadir sus propios plugins.

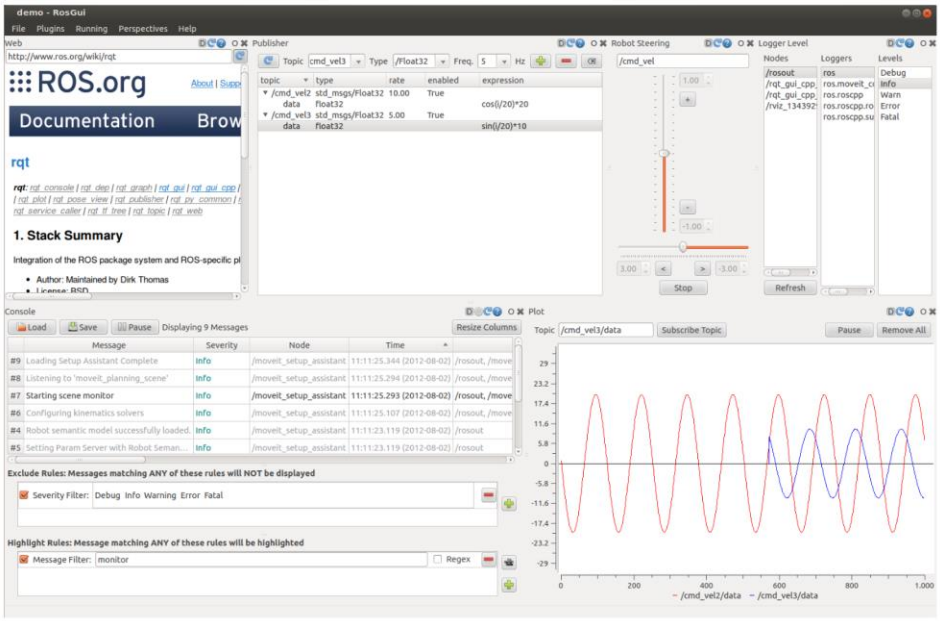


Ilustración 7 RQT

El *rqt_graph* ofrece la visualización en directo del conjunto del sistema ROS, mostrando los nodos y las conexiones entre ellos. De este modo, se puede fácilmente entender el sistema y como está estructurado.

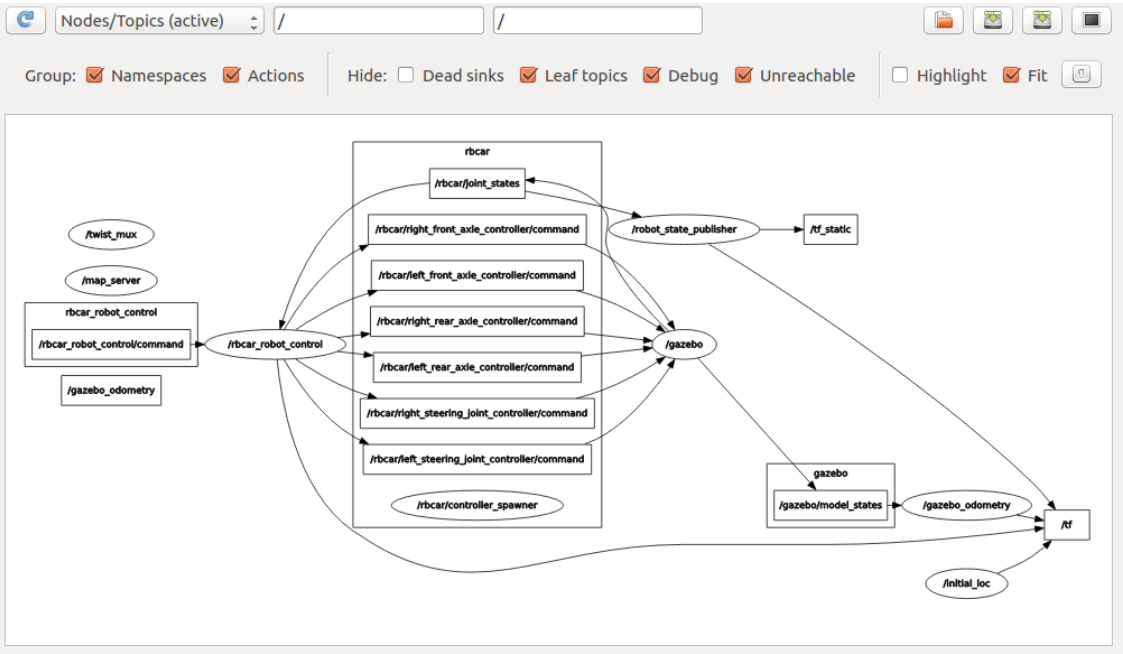


Ilustración 8 RQT graph

Para monitorizar y usar los topics, se tienen los plugins *rqt_topic* y *rqt_publisher*. El primero, permite monitorizar y analizar el número de topics que se están publicando en el sistema. El otro (ilustración 9), permite publicar mensajes a cualquier topic, facilitando así la experimentación en el sistema.

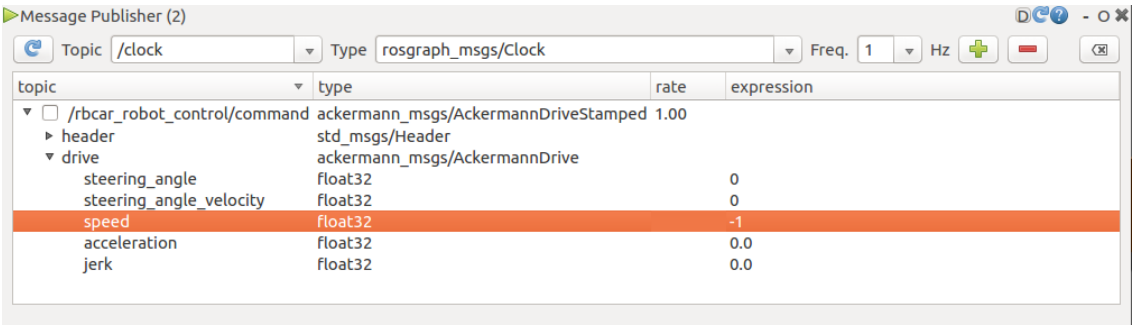


Ilustración 9 rqt_publisher

Con el *rqt_plot* se puede visualizar cualquier varible del sistema que varía con el tiempo, como podría ser los encoders, la posición, velocidad o la tensión.

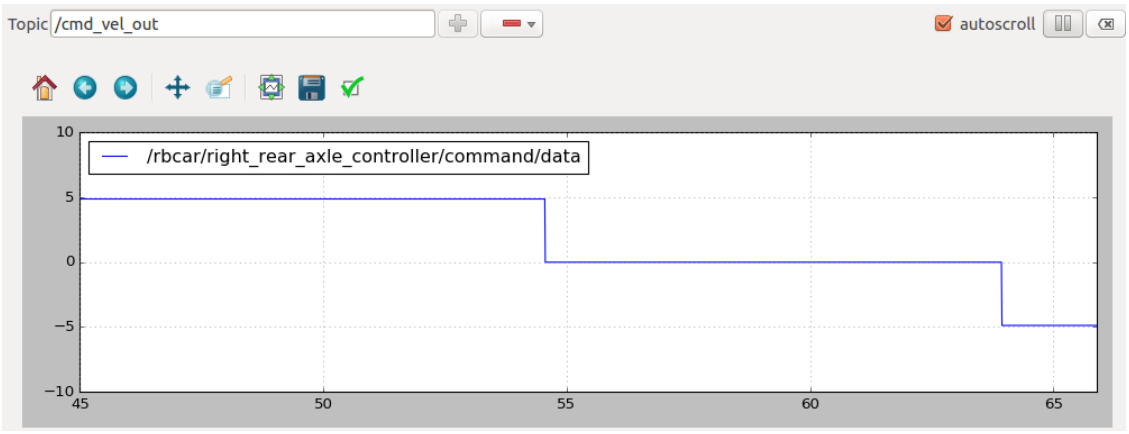


Ilustración 10 rqt_plot

4.3 INTEGRACIÓN CON GAZEBO

ROS ofrece la posibilidad de combinarse e integrarse con otras aplicaciones como Gazebo, moveit! y OpenCV entre otros. El sistema de intercambio de mensajes de ROS permite que la integración con otras aplicaciones. Al igual que ROS, comparten la misma filosofía de código abierto, lo que permite que haya una gran comunidad detrás de estos programas.

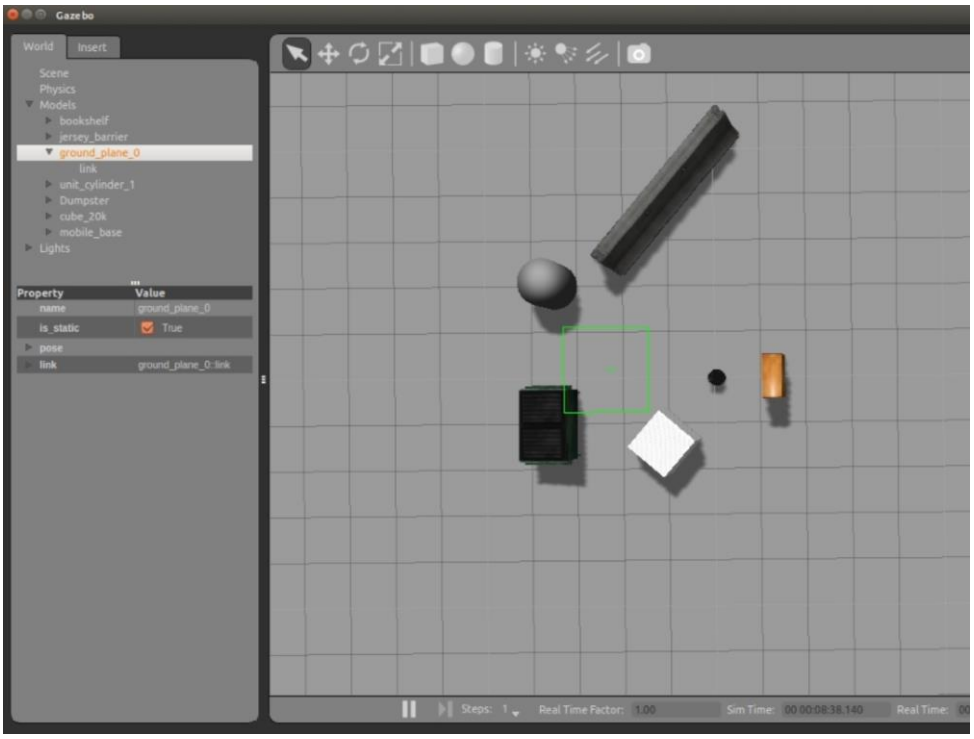


Ilustración 11 Gazebo

Gazebo [6] es un simulador 3D que permite analizar algoritmos, diseñar robots, ejecutar pruebas de regresión y entrenar sistemas de inteligencia artificial usando escenarios realísticos. Gazebo tiene la capacidad de simular poblaciones de robots en entornos complejos, tanto interiores como al aire libre. Sus principales características son las siguientes:

- Sus principales funcionalidades se centran en el sector la robótica, por lo que sus funcionalidades son muy específicas y se puede lograr un gran nivel de detalle en la ejecución de las simulaciones.
- Es multiplataforma, de modo que puede ser usado en Unix como en Windows.
- Motor de físicas robusto que permite realizar simulaciones de manera muy realística y parecida a la realidad.
- Alta calidad en los gráficos, que permite desarrollar y visualizar los modelos con un gran nivel de detalle.
- Programación sencilla e intuitiva, por lo que no se requiere unos grandes conocimientos previos para poder entender su funcionamiento. Además, varias acciones se pueden llevar a cabo mediante la interfaz gráfica.
- Es de software libre, lo que además permite tener una comunidad de desarrollo e intercambio de conocimiento.

5 LENGUAJES DE PROGRAMACIÓN Y APLICACIONES

5.1 XML

XML [7], del inglés *eXtensible Markup Language* es un meta-lenguaje que permite definir lenguajes de marcas. Desarrollado por el *World Wide Web Consortium* (W3C), es utilizado para almacenar datos de manera legible. Proviene del lenguaje SGML y permite definir la gramática de lenguajes específicos para estructurar documentos grandes. A diferencia de otros lenguajes, XML da soporte a bases de datos, siendo útil cuando varias aplicaciones deben comunicarse entre sí o integrar información.

XML es una tecnología sencilla que tiene a su alrededor otras que la contemplan que la hacen mucho más grande, con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

Un documento XML es un texto jerárquico legible por una máquina, que además puede ser leído y comprendido por un humano con un editor de texto básico.

Sus principales ventajas son las siguientes:

- Fácil de leer y comprender para los usuarios humanos, gracias a estar empaquetado y al nombre de las etiquetas.
- Es muy flexible y adaptable a las distintas necesidades, debido a que resulta relativamente sencillo la adición de nuevas etiquetas, de modo que se puede utilizar sin complicación alguna.
- Resulta sencillo de usar y ser leído por otros programas, a consecuencia de su sencilla estructura.
- Es fácil de convertir a otras representaciones o a otros formatos, gracias a que es fácil de interpretar ye empaquetar.
- Contiene muchos estándares y herramientas, que permiten en desarrollo de funcionalidades muy específicas logrando un nivel de detalle muy alto.
- Está muy extendido a nivel mundial por lo que también resulta mantenido, por lo que existe toda una comunidad detrás que permite el intercambio de conocimiento.

5.1.1 Estructura de un documento XML:

El lenguaje XML contiene la información de una manera jerárquica y empaquetada, a fin de poder ser legible y comprensible para las máquinas, además que por los humanos. Un documento XML está compuesto por un prólogo y el cuerpo, siendo el cuerpo el lugar donde se define la estructura jerárquica mediante las etiquetas.

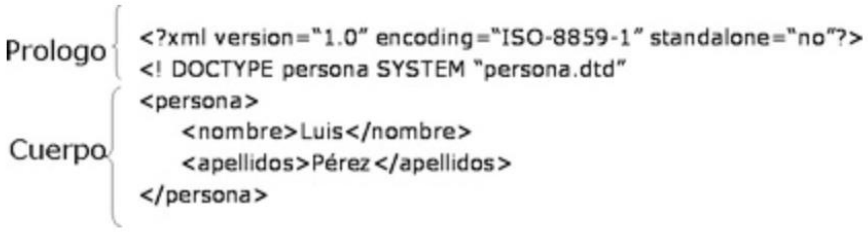


Ilustración 12 Estructura documento XML

En el **prólogo** se define el tipo de documento, la versión, el autor entre otros; aunque no es obligatorio que exista en el documento.

En el **cuerpo** se definen los Elementos, que son las unidades básicas para definir las estructuras jerárquicas. Cada Elemento tiene una etiqueta para poder ser identificado. Además, cada Elemento tiene puede contener uno o varios atributos con sus nombres y sus valores relacionados. También se pueden añadir entidades predefinidas con el objetivo de llevar a cabo una acción específica que no puede interpretar el procesador XML.

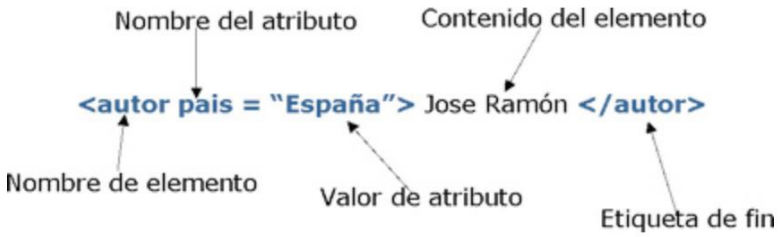


Ilustración 13 Ejemplo de Elemento en XML

Por último, se pueden añadir **comentarios** al documento, cuya sintaxis debe empezar por <!—y debe terminar por -->.

5.2 URDF

El formato unificado de descripción del robot, en inglés *Universal Robot Description Format* (URDF) [8] es una especificación XML que sirve para describir y diseñar un robot. La principal limitación es que solo se pueden representar estructuras comprendidas en el mismo árbol, descartando así otros posibles robots en paralelo. Del mismo modo, en esta especificación solo se pueden representar eslabones rígidos conectados a las articulaciones, por lo que los elementos flexibles no están comprendidos. Esta especificación representa:

- La descripción cinemática y dinámica del robot.
- La representación visual del robot.
- El modelo de colisiones del robot.

La estructura resultante en esta especificación es una serie de links, eslabones, que van siendo unidos a través de joints, articulaciones.

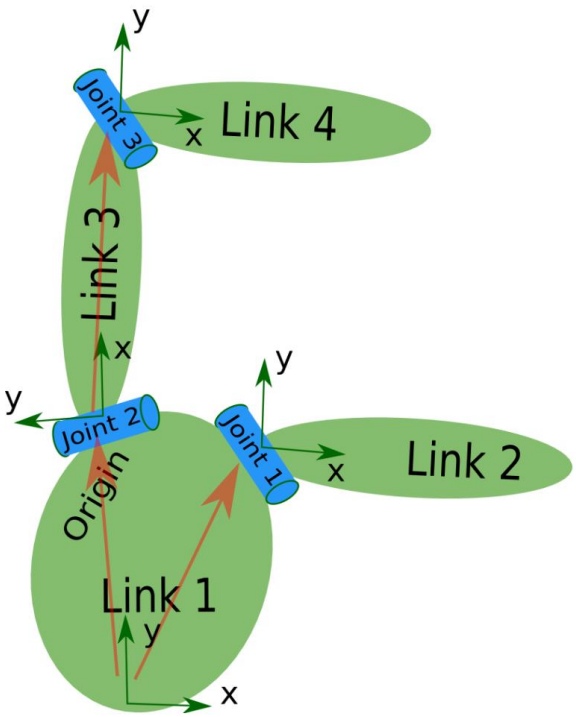


Ilustración 14 Representación de los elementos en un modelo URDF

Como se puede observar en la ilustración 14, la descripción más básica de un robot consiste en un conjunto de links (eslabones) y de joints (articulaciones). Las articulaciones permiten conectar todos los elementos del robot entre sí, y de este modo, formar la estructura completa. Además, también se pueden añadir y configurar otros elementos del robot como son los sensores, los actuadores en las articulaciones y el estado del modelo en un momento determinado.

La estructura URDF del robot de la ilustración 14 debería asemejarse a este código:

```
<robot name="pr2">
  <link1> ... </link1>
  <link2> ... </link2>
  <link3> ... </link3>
  <link4> ... </link4>
  <joint1>  ....  </joint1>
  <joint2>  ....  </joint2>
  <joint3>  ....  </joint3>
</robot>
```

5.2.1 El elemento *link:*

El elemento *link* [9] es el que hace referencia a cada eslabón o parte sólida del robot, debido a las limitaciones del leguaje URDF siempre tendrá que ser rígido y no podrá ser flexible. Un *link* está compuesto por tres partes, una parte visual, que es la relacionada a todos los elementos estéticos; una parte de colisión, que define los límites de la pieza; y un elemento de inercia, que se usa para establecer la inercia del elemento de cara a realizar las simulaciones. (ilustración 15)

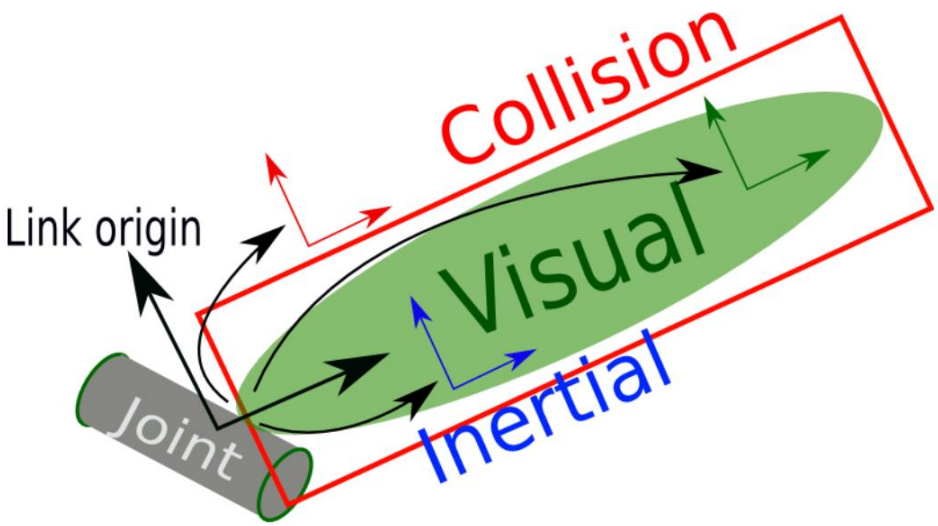


Ilustración 15 Estructura de un link

Atributos:

name (requerido): es el nombre del eslabón. Los demás elementos se refieren al eslabón con este nombre.

Elementos:

<visual> (opcional):

Es la representación gráfica del eslabón, se puede establecer como geometría o con una malla 3D externa.

name (opcional)

Establece un nombre a esta parte del eslabón

<origin> (opcional, si no se especifica será el equivalente a la matriz identidad)

Establece la referencia del elemento visual respecto a la referencia del eslabón.

xyz (opcional, por defecto (0,0,0))

Representa el desplazamiento en cada coordenada.

rpy (opcional, si no se especifica será la matriz identidad)

Representa el ángulo de giro sobre cada eje.

<geometry> (requerido)

Define la geometría del elemento visual, puede una de las siguientes:

<box>

Define un prisma como geometría, se puede establecer la longitud de sus costados mediante el atributo *size.*

<cylinder>

Define cilindro como geometría, se puede establecer su radio y la altura mediante los atributos *radius* y *lenght* respectivamente.

<sphere>

Define una esfera como geometría, se puede establecer el radio mediante el atributo *radius.* El origen está situado en el centro

<mesh>

Se importa una figura en tres dimensiones, tiene un atributo *scale* que permite variar el tamaño según la necesidad. Se pueden importar fichero en formato .dae o .stl.

<material> (opcional)

Especifica el color o textura que se va a enseñar.

name (opcional)

Nombre de referencia al color o a la textura.

<color> (opcional)

Se establece el color mediante una combinación rgba; rojo, verde, azul, opacidad; comprendido cada uno entre 0 y 1

<texture> (opcional)

Un fichero externo conteniendo la textura, debe ser formato .jpg, .png o .bmp

<inertial> (opcional):

Es el elemento que contiene la información respecto a las propiedades de colisión del eslabón, comparte varios atributos con la parte visual.

name (opcional)

Establece un nombre a esta parte del eslabón

<origin> (opcional, si no se especifica será el equivalente a la matriz identidad)

Especifica la referencia de la parte de la colisión del eslabón, al igual que la parte visual contiene los atributos *xyz* y *rpy* para definir su posición y orientación.

<geometry> (requerido)

Define la geometría del elemento de colisión, puede una de las que se han mencionado en la parte visual: box, cylinder, sphere o radius.

<inertial> (opcional):

Es el elemento que contiene la información respecto a las propiedades inerciales del eslabón.

<origin> (opcional, si no se especifica será el equivalente a la matriz identidad)

Especifica la referencia del origen de los momentos de inercia, al igual que la parte visual y la de colisión contiene los atributos *xyz* y *rpy* para definir su posición y orientación.

<mass> (opcional)

Define la masa del eslabón en gramos

<inertia> (opcional)

Matriz 3x3 del rotacional de inercia, al ser simétrico solo se pueden definir los atributos ixx, ixy, ixz, iyy, iyz e izz.

5.2.2 El elemento *joint:*

El elemento joint [10] es el que permite unir eslabones y definir el modo como están conectados formando así las articulaciones. También se establecen los límites del movimiento de unión.

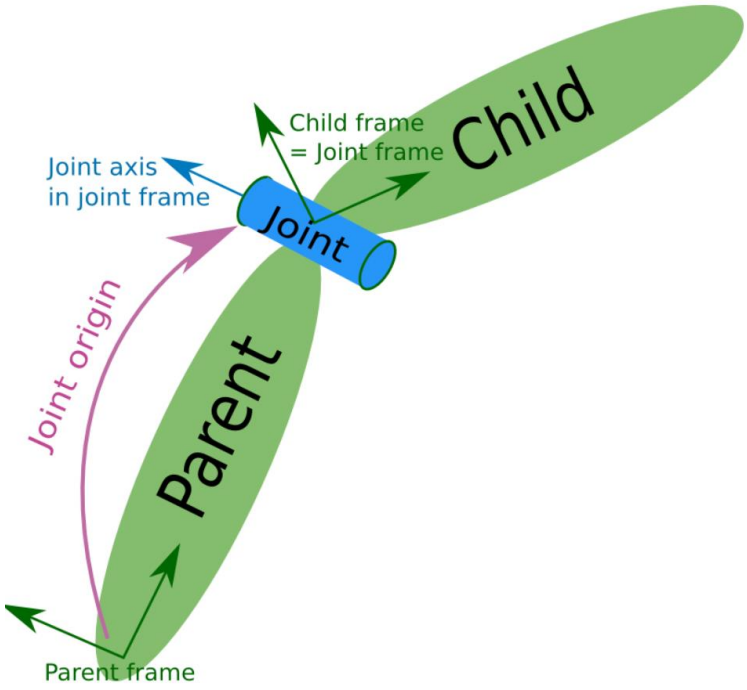


Ilustración 16 Ejemplo de articulación

Existen varios tipos de articulaciones (tabla 4), para poder ser adaptadas a las necesidades de cada proyecto según convenga:

revolute	Permite la rotación sobre sus ejes con límites de movimiento superior e inferior
continous	Permite la rotación sobre sus ejes sin límites
prismatic	Permite el deslizamiento sobre sus ejes con límites superior e inferior
fixed	No se permite el movimiento, todos sus grados de libertad están bloqueados
floating	Permite el movimiento de manera libre sobre sus seis grados de libertad
planar	Se permite el movimiento de manera libre en un plano perpendicular al eje

Tabla 4 Tipos de articulaciones

Es importante comprender de manera clara los conceptos *parent* (padre) y *child* (hijo) en las articulaciones (ilustración 16). El primero hace referencia al elemento al que la articulación le pertenece, mientras que el segundo es el que pertenece a la articulación, es decir, se respeta el orden jerárquico del robot. Por ejemplo, en un robot humanoide, el eslabón padre sería el tronco y el hijo serían los brazos.

Atributos:

name (requerido):

es el nombre de la articulación. Los demás elementos se refieren a la articulación con este nombre.

type (requerido):

tipo de articulación, tiene que ser una de las seis que se han mencionado en la tabla 4.

Elementos:

<origin> (opcional, si no se especifica será el equivalente a la matriz identidad)

Es la transformada que va desde el eslabón padre al eslabón hijo. El origen está situado en el origen del eslabón hijo.

xyz (opcional, por defecto (0,0,0))

Representa el desplazamiento en cada coordenada.

rpy (opcional, si no se especifica será la matriz identidad)

Representa el ángulo de giro sobre cada eje.

<parent> (requerido)

Establece el eslabón padre.

<child> (requerido)

Establece el eslabón hijo.

<axis> (opcional)

Vector *xyz* que especifica los ejes de la articulación.

<limit> (opcional)

Elemento para establecer los límites, se pueden configurar estos atributos:

lower: se especifica el límite inferior.

upper: se especifica el límite superior.

effort: se especifica el máximo esfuerzo de la articulación.

velocity: se especifica la máxima velocidad que puede alcanzar la articulación.

<calibration> (opcional)

Se establecen las referencias para futuras calibraciones.

rising: cuando la articulación se mueve en dirección positiva, se lanzará un flanco ascendente.

rising: cuando la articulación se mueve en dirección negativa, se lanzará un flanco descendente.

<dynamics> (opcional)

Se establecen los coeficientes de **damping** y **friction** en la articulación.

<mimics> (opcional)

Se usa para replicar el comportamiento de una articulación ya definida:

lower: el nombre de la articulación a replicar.

multiplier: se establece el multiplicador para el movimiento de la articulación.

offset: se establece un desplazamiento sobre el parámetro anterior.

<safety_controler> (opcional)

Establece los límites de velocidad y esfuerzo en el controlador del robot.

soft_lower_limit: establece el límite inferior para empezar a limitar la posición.

soft_upper_limit: establece el límite superior para empezar a limitar la posición.

k_position: relación entre la posición y el límite de velocidad.

k_velocity: relación entre el esfuerzo y el límite de velocidad.

5.2.3 Transmisiones URDF:

El elemento *transmission* [11] (<transmision>) es una extensión del lenguaje para la descripción del robot URDF que es usado para describir la relación entre un actuador y una articulación. Esto permite modelizar conceptos como la relación de transmisión o la cinemática paralela. El elemento transmisión transforma esfuerzos variables de manera que su potencia permanece constante. Múltiples actuadores deben de estar conectados a múltiples articulaciones a través de transmisiones complejas.

Atributos:

name (requerido):

Es el nombre de la transmisión.

Elementos:

<type> (requerido una aparición)

Especifica el tipo de transmisión

<joint> (requerido una o más apariciones)

Se especifica el nombre de la articulación a la cual se va a conectar mediante el atributo **name**.

<hardwareInterface> (requerido una o mas apariciones)

Establece un espacio articular que pueda ser soportado.

<actuator> (requerido una o más apariciones)

Tipo de actuador que se va a conectar a la articulación, se especifica mediante el atributo **name**.

<mechanicalReduction> (opcional)

Especifica la reducción mecánica que se aplica a la transmisión.

<hardwareInterface> (requerido una o mas apariciones)

Establece un espacio articular que pueda ser soportado.

5.2.4 El elemento *gazebo*:

El elemento <gazebo> [12] es una extensión del lenguaje URDF usado para especificar las propiedades específicas que permiten la simulación en Gazebo. Acepta especificar propiedades propias del formato SDF (*System Description File*) que no están incluidas en el formato URDF. Ningún elemento <gazebo> es estrictamente necesario debido a que ya se establecen los valores por defecto. Existen tres tipos de elementos <gazebo>, para robot, para link y para joint.

Elementos

<robot> (requerido)

<gazebo> (opcional)

<static> (opcional)

Si es true, es modelo permanecerá inmóvil, en caso contrario, la dinámica del modelo es simulada.

<link> (requerido)

<gazebo> (opcional)

<material> (opcional)

Material visual del elemento.

<gravity> (opcional)

Si es true se usa la gravedad.

<dampingFactot> (opcional)

Tiempo de caída de la velocidad.

<maxVel> (opcional)

Máximo contacto con el término de corrección de velocidad.

<minDepth> (opcional)

Profundidad mínima antes de que se aplique la corrección de contacto.

<mu1> (opcional)

Coeficiente de fricción en una determinada dirección.

<mu2> (opcional)

Coeficiente de fricción en una determinada dirección.

<fdir1> (opcional)

Vector que especifica la dirección de mu1.

<kp> (opcional)

Rigidez del engranaje.

<kd> (opcional)

Elasticidad del engranaje.

<selfCollide> (opcional)

Si es true, la articulación puede chocar con otras articulaciones.

<maxContacts> (opcional)

Número de contactos permitidos entre dos entidades.

<laserRetro> (opcional)

Valor de intensidad del sensor laser.

<joint> (requerido)

<gazebo> (opcional)

<stopCfm> (opcional)

Limitación de suma de fuerzas en la articulación.

<stopErp> (opcional)

Parámetro de reducción de errores.

<provideFeedback> (opcional)

Permite publicar los datos de torsión si es verdadero.

<ImplicitSpringDamper> (opcional)

Si es verdadero se utiliza la amortiguación cfm.

<cfmDamping> (opcional)

Coeficiente de amortiguación cfm.

<fudgeFactor> (opcional)

Controla el exceso de potencia del motor en las articulaciones.

5.2.4.1 Compatibilidad de plugins de Gazebo con el modelo URDF:

Gazebo ofrece la posibilidad de añadir más funcionalidades que pueden ser emparejadas con mensajes y servicios de ROS. Los plugins se pueden añadir a cualquier elemento del URDF, en función del caso de uso y de la función que le se pretenda dar. Los plugins más populares de Gazebo son las cámaras, el Kinect, sensor laser, sensor IMU entre muchos otros.

5.3 XML XACRO

XACRO es una extensión del lenguaje XML que permite construir archivos XML de manera más corta y más legible. Su principal característica es la posibilidad de crear MACROs, que permiten empaquetar el código, y así evitar escribir líneas de manera redundante. También permite parametriza las entidades que se definen en el código, lo que a consecuencia repercute en la modularidad, es decir, a partir de establecer unos parámetros u otros en una MACRO se pueden generar elementos para ser adaptados a las distintas situaciones que se planteen.

En las siguientes líneas se puede apreciar como ejemplo la estuctura de una MACRO dentro de un archivo .urdf.xacro:

```
<xacro:macro     name="pr2_caster"     params="suffix     *origin     **content
      **anothercontent">
  <joint name="caster_${suffix}_joint">
    <axis xyz="0 0 1" />
  </joint>
  <link name="caster_${suffix}">
    <xacro:insert_block name="origin" />
    <xacro:insert_block name="content" />
    <xacro:insert_block name="anothercontent" />
  </link>
</xacro:macro>
```

```
<xacro:pr2_caster suffix="front_left">
  <pose xyz="0 1 0" rpy="0 0 0" />
  <container>
    <color name="yellow"/>
    <mass>0.1</mass>
  </container>
  <another>
    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
    </inertial>
  </another>
</xacro:pr2_caster>
```

Además de empaquetamiento de código, XACRO ofrece otras utilidades, como sería la posibilidad de definir propiedades, poder usar expresiones matemáticas y permite el uso de bloques condicionales.

5.4 ACHIVOS LAUNCH DE ROS

Los ficheros .launch son muy comunes en ROS, son usados tanto por usuarios como por desarrolladores. Ofrecen una manera óptima de inicializar múltiples nodos y el master [13]. El comando de línea que se utiliza para abrir los ficheros .launch es *roslaunch*, donde se le puede especificar el paquete en el que está incluido el archivo o incluir directamente la ruta hacia el mismo.

Los archivos .launch usan un formato específico de XML. Se puede ubicar en cualquier ubicación dentro del directorio de se paquete de ROS correspondiente, pero normalmente se sitúan dentro de una carpeta “Launch”.

Los contenidos dentro de un archivo .launch están siempre entre un par de etiquetas <launch> y </launch>, como se puede ver en el siguiente ejemplo, donde se inicializan dos nodos:

```
<launch>
    <node pkg="tf" type="static_transform_publisher" name="initial_loc"
        args="25 25 0 0 0 0 /map /odom 2" output="screen"/>


    <node name="map_server" pkg="map_server" type="map_server"
        args="$(find rbcar_gazebo)/maps/map1.yaml" output="screen">
        <param name="frame_id" value="map"/>
```

```
    </node>


</launch>
```

Se deben de definir una serie de atributos en la inicialización de cada nodo:

pkg: paquete asociado al nodo que se va a iniciar.

type: nombre del nodo que se va a ejecutar.

name: es posible sobrescribir el nombre del nodo mediante este argumento.

respawn: si es igual a true, el nodo se va a reiniciar si por algún motivo se cierra.

required: si es igual a true, se cerrarán todos los nodos asociados a este si por algún motivo se cierra.

ns: se puede especificar el *namespace* mediante este argumento.

arg: se pueden utilizar variables locales, a través de la estructura:

<arg name="…" value="…">

Además, también se pueden definir otros elementos mediante etiquetas:

<include>: permite incluir otros ficheros .launch en el fichero actual.

<param>: define un parámetro para que pueda ser utilizado.

<rosparam>: permite cargar parámetros definidos en un fichero YAML para que puedan ser usados.

5.5 SKETCHUP

Sketchup es una plataforma CAD que permite crear y editar conceptos 2D y 3D. Esta aplicación permite a los usuarios crear modelos 3D de interiores, muebles, modelos, paisajes y más. Su punto más fuerte es su facilidad de uso, gracias a al conjunto de herramientas de calidad que posee.

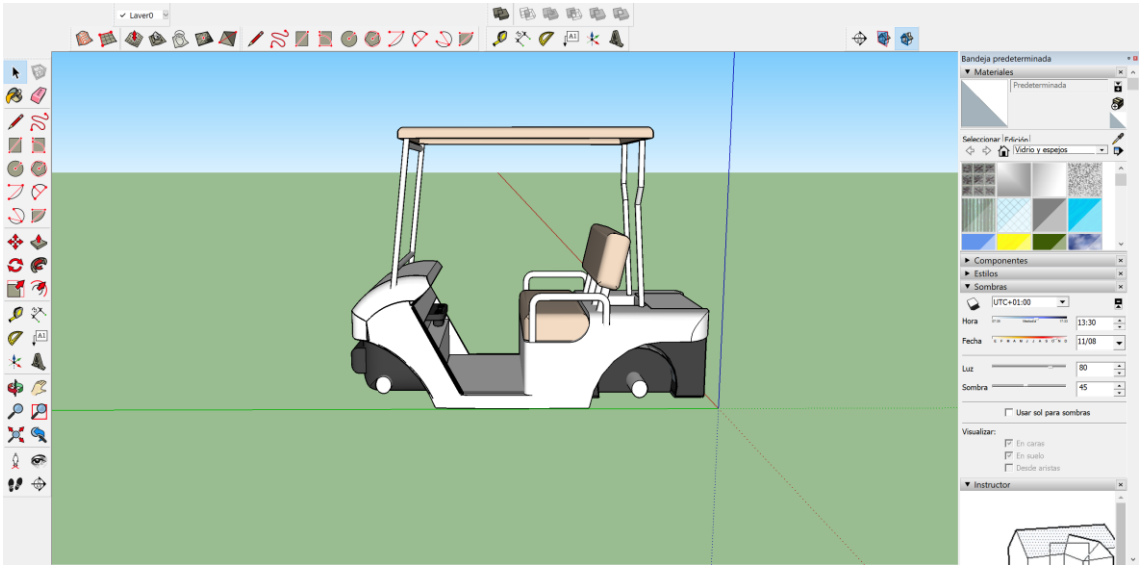


Ilustración 17 Sketchup

Los principales puntos fuertes de este software son los siguientes:

- Posibilidad de crear y editar modelos 2D y 3D.
- Sencillo de aprender y comprender.
- Se puede lograr un gran nivel de detalle en los desarrollos.
- Posee una versión gratuita bastante completa.
- Existe una comunidad de soporte e intercambio de conocimiento.

6 IMPLEMENTACIÓN

Para llevar a cabo la modelización del vehículo y así poder lograr los objetivos establecidos, se ha dividido la fase de desarrollo en distintas sub etapas, que ayudan a definir cada proceso y a ejecutar cada desarrollo de manera más ordenada. Las etapas que se han definido son las siguientes:

- Simulación de la geometría de Ackermann.
- Creación del modelo en 3D.
- Ajuste del modelo 3D a la geometría de Ackermann.
- Adición del sensor Velodyne.
- Adición de la cámara estéreo.
- Adición del sensor laser.
- Creación de entorno de pruebas.
- Localización del vehículo.
- Simulación del modelo completo.

Al final de esta sección se habrá llevado a cabo la modelización del vehículo. El siguiente paso va a ser realizar las simulaciones para poder evaluar la calidad de la modelización y el comportamiento de las misma. Para ello, en este mismo capítulo también se define un entorno de pruebas que va a servir como referencia en el momento de realizar las simulaciones.

6.1 SIMULACIÓN DE LA GEOMETRÍA DE ACKERMANN

Uno de los motivos por los que se ha optado por ROS es por la inmensa comunidad permite la posibilidad de compartir conocimiento, intercambio de información y reutilización de código. Dentro de la comunidad de ROS-Gazebo existen varios paquetes accesibles de manera libre que contienen controladores que respetan la geometría de Ackermann.

Dentro de la comunidad de ROS-Gazebo existen varios paquetes accesibles de manera libre que contienen controladores que respetan la geometría de Ackermann.

Se ha llevado a cabo una labor de búsqueda y análisis por la red y se encontrado el paquete *ackermann_vehicle* [14], que contiene los paquetes de ROS que permiten simular un vehículo con la geometría de Ackermann en sus ruedas delanteras.

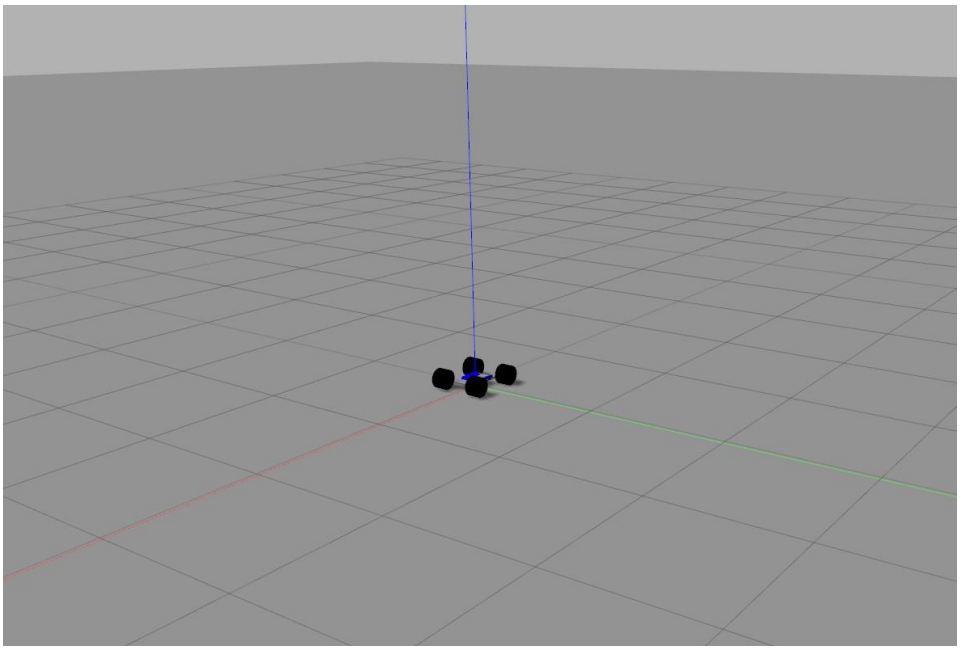


Ilustración 18 Paquete ackermann_vehicle

Tras evaluar varios parámetros como las dimensiones del chasis, de las ruedas y las masas del vehículo, se concluye que es un modelo que funciona muy bien en vehículos de pequeño tamaño, pero que al aumentar del modelo se dan algunos problemas, por lo que no es el paquete más óptimo para utilizar.

Por otro lado, se tiene el paquete rbcar_sim [15], que contiene una serie de paquetes que permiten ejecutar la simulación de un vehículo con la geometría de Ackermann. En este caso la diferencia es que este paquete se ha diseñado para llevar a cabo simulaciones el RB-CAR [16], un robot móvil para la navegación autónoma, hecho por Robotnik.

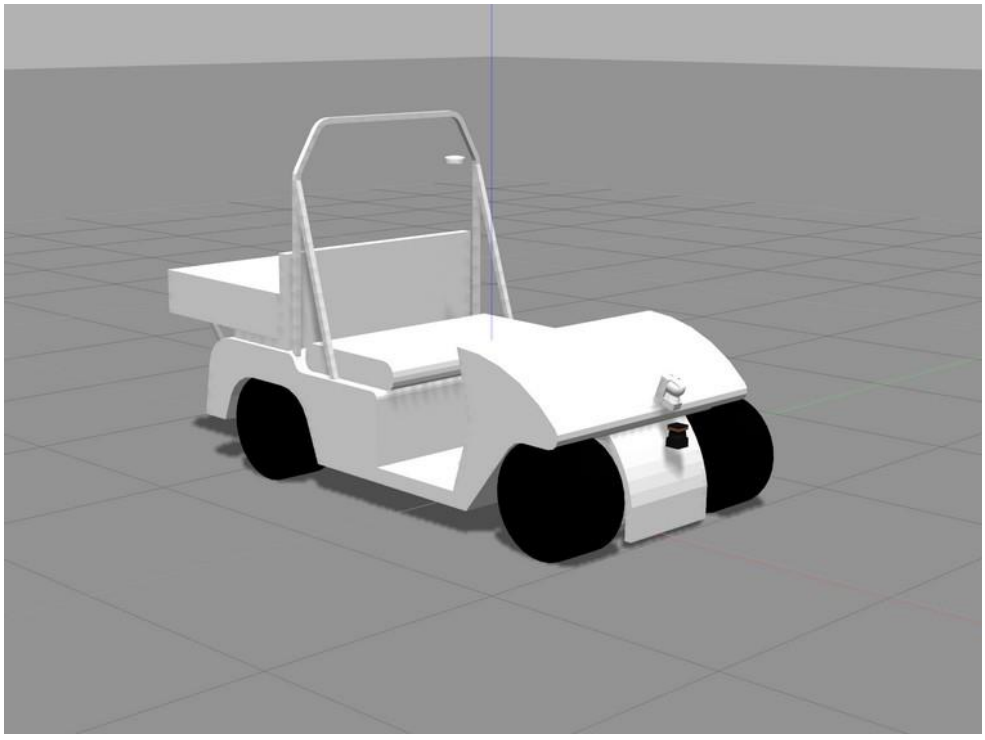


Ilustración 19 RBCar de Robotnik

Tras evaluar el modelo y comprobar que se está tratando de dimensiones semejantes al iCab que se pretende simular, se considera que este puede ser un buen controlador con el que se puede continuar el proyecto.

6.2 CREACIÓN DEL MODELO EN 3D

Para llevar a cabo esta tarea, se ha utilizado el programa Sketchup que permite la edición de modelos en tres dimensiones. Además, contiene una biblioteca de modelos ya predefinidos. Así pues, se ha seleccionado uno, *Golf Cart*, de esta biblioteca para editarlo y adaptarlo al vehículo que se pretende simular:

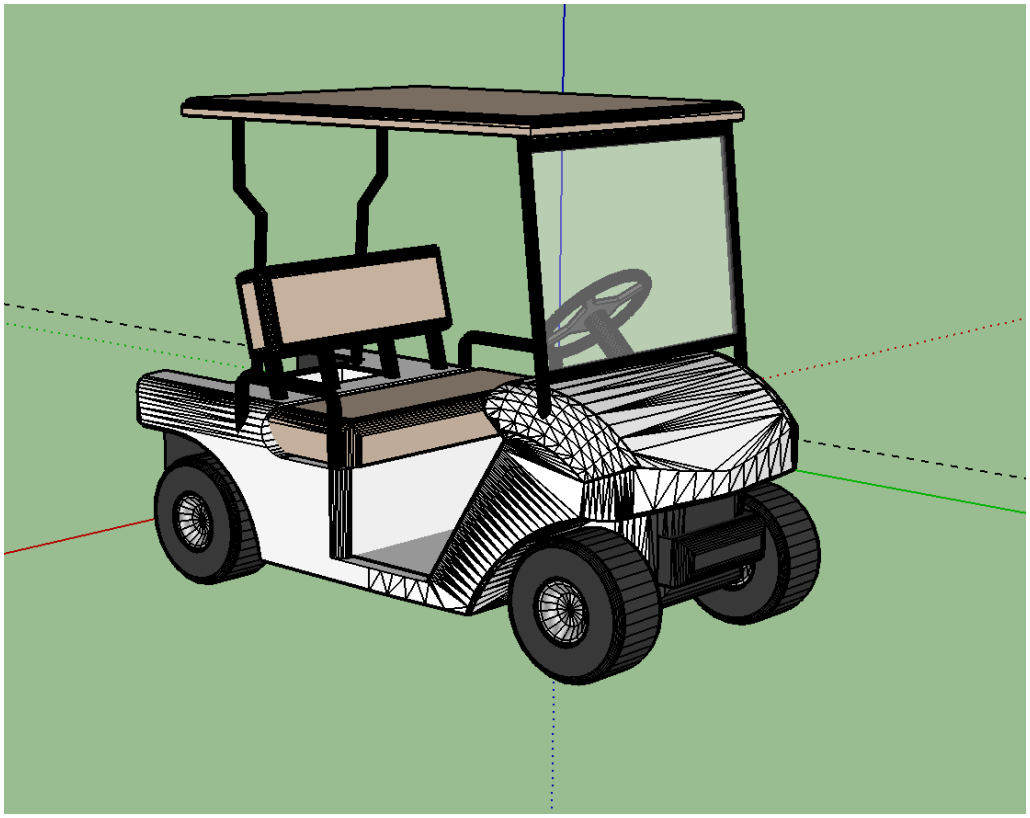


Ilustración 20 Golf Cart

Sobre este vehículo se han realizado varias acciones, primero se han suavizado las texturas y después se han eliminado los elementos que no son propio del vehículo que pretendemos simular, como serían el volante, los pedales y el cristal. También se eliminan las ruedas porque se van a utilizar las que vienen con el controlador visto en el apartado anterior, para así respetar visualmente la geometría de Ackermann.

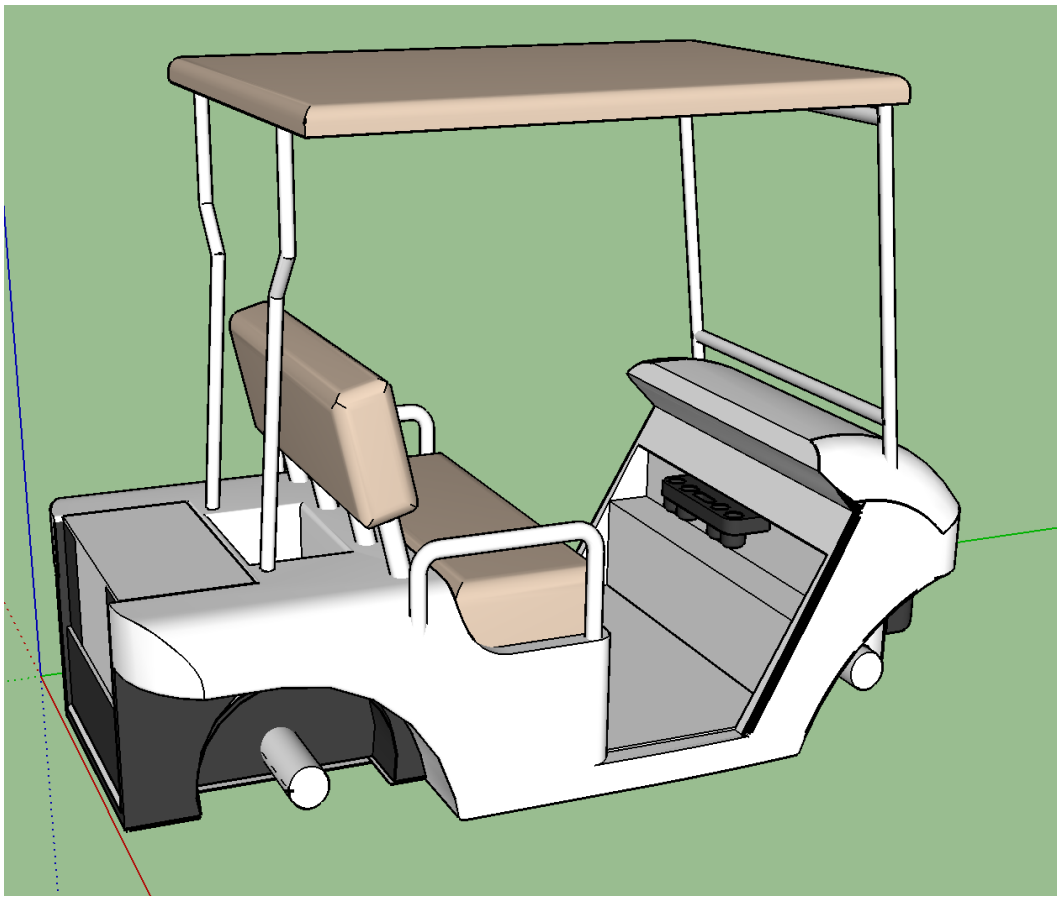


Ilustración 21 Golf Cart tras primeros desarrollos

Después, debido a que las ruedas del vehículo que se pretende modelizar son mayores que las que venían por defecto en el modelo, se tiene que hacer una extrusión en la parte del eje trasero del vehículo para que así los neumáticos no colisionen con el chasis.

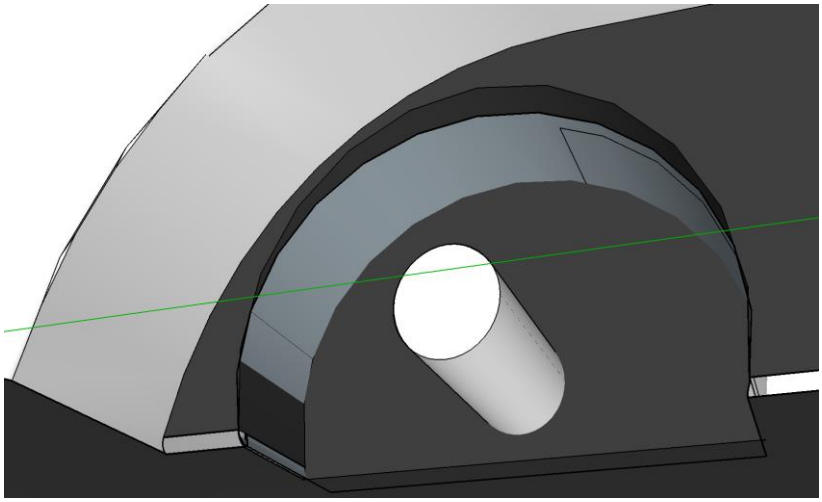


Ilustración 22 Parte extruída en la rueda trasera izquierda

Finalmente, se ha adaptado el modelo a las dimensiones reales del vehículo, obteniendo así el modelo final.

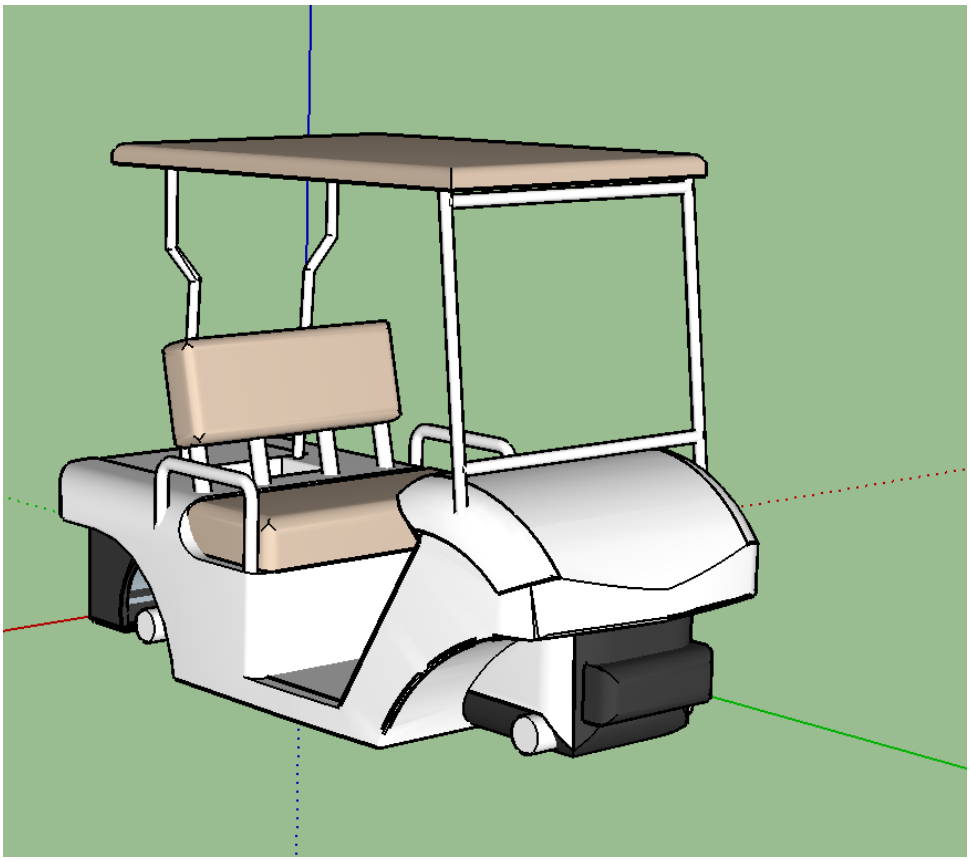


Ilustración 23 Modelo final

El modelo, para que pueda ser visualizado en Gazebo, se exporta en formato Collada, con el nombre de mod8.dae.

6.3 AJUSTE DEL MODELO 3D AL MODELO DE ACKERMANN

6.3.1 Ajuste del chasis:

Una vez se ha seleccionado un controlador para gobernar el vehículo y además se tiene un modelo tridimensional para representarlo, el siguiente paso es ensamblar ambas partes y conseguir que funcionen correctamente como conjunto.

De este modo, para poder lograrlo se edita el fichero rbcar_base.urdf.xacro, que es el que contiene toda la información relativa al chasis del vehículo. Consiste en un archivo .urdf.xacro que define una .macro para representar el cuerpo del coche.

Para ello, se representa un eslabón de referencia, base_footprint, que consiste en un cubo de tamaño prácticamente despreciable al que se le pegará el chasis del coche.

```
<link name="base_footprint">
```

```
      <visual>
          <origin xyz="0.5 0 0" rpy="0 0 0" />
          <geometry>
             <box size="0.001 0.001 0.001" />
          </geometry>
      </visual>
</link>
```

La articulación base_footprint_joint es la unión fija entre la base de referencia y lo que es el chasis propiamente dicho, representado en el eslabón base_link.

```
<link name="base_link">
    <inertial>
      <mass value="24.0" />
      <origin xyz="0 0 0.3" />
      <inertia  ixx="1.391" ixy="0.004"  ixz="0.0"  iyy="6.853"  iyz="0.0"
       izz="6.125" />
    </inertial>
    <visual>
       <origin xyz="-1.29 0.65 0.04" rpy="0 0 ${-degrees_90}"/>
       <geometry>
            <mesh
      filename="package://rbcar_description/meshes/bases/mod8.dae"/>
       </geometry>
       <material name="darkgrey">
         <color rgba="0.1 0.1 0.1 1"/>
       </material>
    </visual>
    <collision>
       <origin xyz="-1.29 0.65 0.04" rpy="0 0 ${-degrees_90}"/>
       <xacro:if value="${hq}">
             <geometry>
            <mesh
      filename="package://rbcar_description/meshes/bases/mod8.dae"/>
            </geometry>
       </xacro:if>
       <xacro:unless value="${hq}">
       <geometry>
             <box size= "2.36 0.85 1.23"/>
```

```
       </geometry>
       </xacro:unless>
    </collision>
</link>
```

En esta parte del código se establece el conjunto del chasis al completo, primero se establece la masa que tendrá en la simulación y los momentos de inercia.

```
    <inertial>
      <mass value="24.0" />
      <origin xyz="0 0 0.3" />
      <inertia  ixx="1.391" ixy="0.004"  ixz="0.0"  iyy="6.853"  iyz="0.0"
       izz="6.125" />
    </inertial>
```

A continuación, se importa la malla que va a representar la parte visual del eslabón y se establece su posición. El modelo Collada que se va a importar tiene su origen de coordenadas en su esquina inferior izquierda, por lo que sus coordenadas se han ajustado para que el modelo tridimensional se ajuste a la geometría de las ruedas, tanto las que hacen referencia a la posición, como las que definen su orientación.

```
       <origin xyz="-1.29 0.65 0.04" rpy="0 0 ${-degrees_90}"/>
```

Del mismo modo, se configura la parte que hace referencia a la colisión del elemento. Se hace en función de un parámetro, hq, que se define cuando se hace la llamada al archivo .macro. En caso de que este parámetro sea verdadero, la parte colisional será exactamente igual que la visual, donde se usará la misma geometría definida por el archivo Collada.

```
<xacro:if value="${hq}">
       <geometry>
            <mesh
      filename="package://rbcar_description/meshes/bases/mod8.dae"/>
       </geometry>
</xacro:if>
```

En caso que el parámetro hq sea falso, se utiliza una geometría en forma de prisma para establecer el elemento colisión. Sus dimensiones se adaptan a la geometría de la parte visual. Esto permite consumir menos recursos de la máquina donde se ejecute, a expensas de perder precisión en el elemento.

```
<xacro:unless value="${hq}">
       <geometry>
            <box size= "2.36 0.85 1.23"/>
```

```
        </geometry>
</xacro:unless>
```

De este modo se constituye la parte del chasis del vehículo, empaquetada en un fichero .xacro.macro. En el archivo rbcar.urdf.xacro, se le llamará de la siguiente forma:

```
<xacro:rbcar_base name="rbcar" publish_bf="true" hq="${hq}" />
```

Donde se tienen dos atributos:

publish_bf: si es verdadero se publica el elemento base_footprint.

hq: parámetro para controlar la calidad del elemento de colisión.

6.3.2 Configuración de las ruedas::

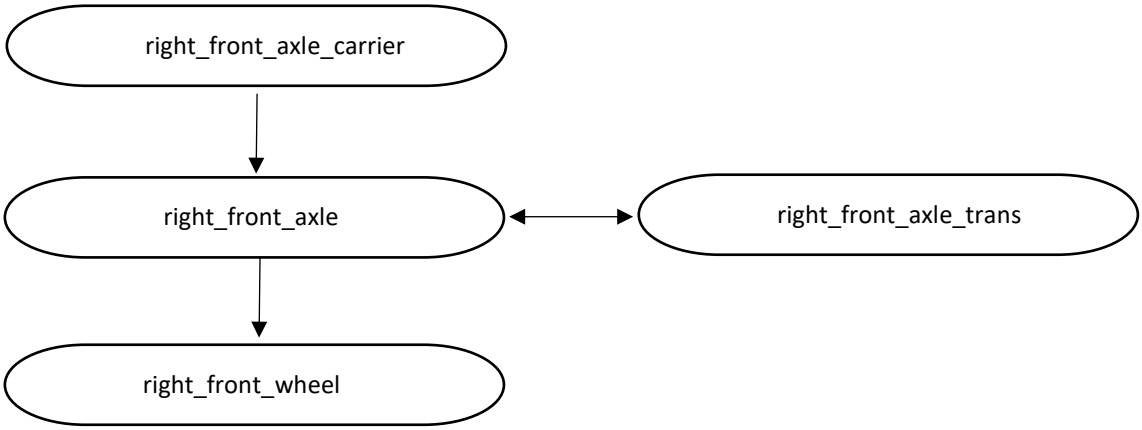
La creación y configuración de las ruedas se hace en el archivo suspension_wheel.urdf.xacro, donde se definen dos macro, front_wheel y rear_wheel, que permiten representar las ruedas del vehículo. Son muy parecidas entre ellas, aunque cada una tiene alguna particularidad que se podrá apreciar seguidamente.

Primero se definen una serie de propiedades comunes para las dos ruedas.

```
<xacro:property name="hub_dia" value="0.235"/>
<xacro:property name="tire_dia" value="0.470"/>
<xacro:property name="tire_width" value="0.2"/>
<xacro:property name="wheelbase" value="1.650"/>
<xacro:property name="hex_hub_dist" value="0.900"/>
<xacro:property name="axle_length" value="0.120"/>
```

Se establece el diámetro de las ruedas, su anchura y su masa. A continuación, la distancia máxima entre ambas ruedas, la distancia del eje al exterior de la rueda y el recorrido de la rueda respectivamente. Todas las demás propiedades definidas en el archivo no se han editado.

También tienen en común la forma de definir las articulaciones y sus amortiguadores, parametrizadas para que se puedan adaptar a la configuración de cada rueda. Primero se crea un elemento de referencia, al que mediante una articulación continua se le une la geometría de la rueda. Mientras, a la unión se le aplica el elemento transmisión para los motores puedan tener tracción sobre la misma. Por ejemplo, en la rueda delantera derecha la estructura es la siguiente.



De los cuatro elementos, tres de ellos son iguales en todas las ruedas, es decir, el _axle_carrier, el _axle y el axle_trans.

```
<link name="${lr_prefix}_${fr_prefix}_axle_carrier">
    <visual>
        <origin xyz="0 0 0"/>
        <geometry>
          <cylinder radius="0.1" length="0.1"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0"/>
        <geometry>
          <cylinder radius="0.1" length="0.1"/>
        </geometry>
     </collision >
 </link>


<joint name="${lr_prefix}_${fr_prefix}_axle" type="continuous">
      <parent link="${lr_prefix}_${fr_prefix}_axle_carrier"/>
      <child link="${lr_prefix}_${fr_prefix}_wheel"/>
      <origin rpy="${degrees_90} 0 ${degrees_180}"/>
      <axis xyz="0 0 -1"/>
      <limit effort="${axle_eff_limit}" velocity="${axle_vel_limit}"/>
      <joint_properties                        damping="${wheel_joint_damping}"
       friction="{wheel_joint_friction}"/>
</joint>
```

```
<transmission name="${lr_prefix}_${fr_prefix}_axle_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${lr_prefix}_${fr_prefix}_axle">
        <hardwareInterface>VelocityJointInterface</hardwareInterface>
      </joint>
      <actuator name="${lr_prefix}_${fr_prefix}_axle_act">
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
</transmission>
```

Respecto a las ruedas, la geometría es igual en todas, tanto la parte visual, la de colisión y la inercia, la única diferencia que existe es el origen de las ruedas traseras, donde, que están ligeramente más separadas entre ellas, exactamente 0,2 metros. Así pues, al definir cada una de las ruedas traseras, se suma 0.1 metros en la coordenada del eje trasero.

```
<link name="${lr_prefix}_${fr_prefix}_wheel">
      <visual>
        <origin xyz="0
                     0
                     ${lr_reflect * (0.1+axle_length - (tire_width / 2 -
                    hex_hub_depth))}"/>
        <geometry>
          <cylinder radius="${tire_dia / 2}" length="${tire_width}"/>
        </geometry>
        <material name="tire_mat"/>
      </visual>

      <collision>
        <origin xyz="0
                     0
                     ${lr_reflect * (0.1+axle_length - (tire_width / 2 -
                    hex_hub_depth))}"/>
        <geometry>
          <cylinder radius="${tire_dia / 2}" length="${tire_width}"/>
        </geometry>
      </collision>

      <xacro:thick_walled_tube_inertial
```

```
          inner_rad="${hub_dia / 2}" outer_rad="${tire_dia / 2}"
          height="${tire_width}" mass="${wheel_mass}"/>
 </link>
```

Por último, la parte que hace referencia a la inercia de la rueda es común en todas, definida mediante un .xacro.macro.

```
<xacro:macro name="thick_walled_tube_inertial"
               params="inner_rad outer_rad height mass">
    <inertial>
      <mass value="${mass}"/>
      <inertia ixx="${(1 / 12) * mass * (3 * (inner_rad * inner_rad +
                   outer_rad * outer_rad) + height * height)}"
               ixy="0" ixz="0"
               iyy="${(1 / 12) * mass * (3 * (inner_rad * inner_rad +
                   outer_rad * outer_rad) + height * height)}"
               iyz="0"
               izz="${mass * (inner_rad * inner_rad +
                   outer_rad * outer_rad) / 2}"/>
    </inertial>
</xacro:macro>
```

Los ruedas están definidas en elementos en formato MACRO dentro de suspension_wheel.urdf.xacro, donde en el fichero rbcar.urdf.xacro se invocan de la siguiente manera:

```
<xacro:front_wheel lr_prefix="left" fr_prefix="front"  lr_reflect="1"
            fr_reflect="1"/>
```

Donde se tienen cuatro atributos:

lr_prefix: "left" o "right", se usa para definir los nombres de los elmentos.

fr_prefix: "front" o "rear", se usa para definir los nombres de los elmentos.

lr_reflect: 1 o 0 en función de si es derecha o izquierda respectivamente, se usa para definir la posición de la rueda.

fr_reflect: 1 o 0 en función de si es delantera o trasera respectivamente, se usa para definir la posición de la rueda.

6.4 ADICIÓN DEL SENSOR VELODYNE

El Velodyne que se ha utilizado es un modelo VLP-16 incluido en el repositorio velo2cam_gazebo. Este repositorio [17], desarrollado por un grupo de investigadores del Intelligent Systems Laboratory de la Universidad Carlos III de Madrid, incluye modelos de Gazebo, plugins y mundos para hacer distintas pruebas y calibraciones.

En el repositorio existen distintos modelos de velodynes, el HDL-32 y el HDL-64, pero finalmente se va a incluir el VLP-16 debido a que es el que requiere menos prestaciones de la máquina donde se ejecute.

Se incluye un plugin, PointCloud2, que va a ser el encargado de realizar y mostrar las nubes de puntos a partir de lo que vaya capturando el sensor. Dichos resultados podrán ser vistos mediante el Rviz.

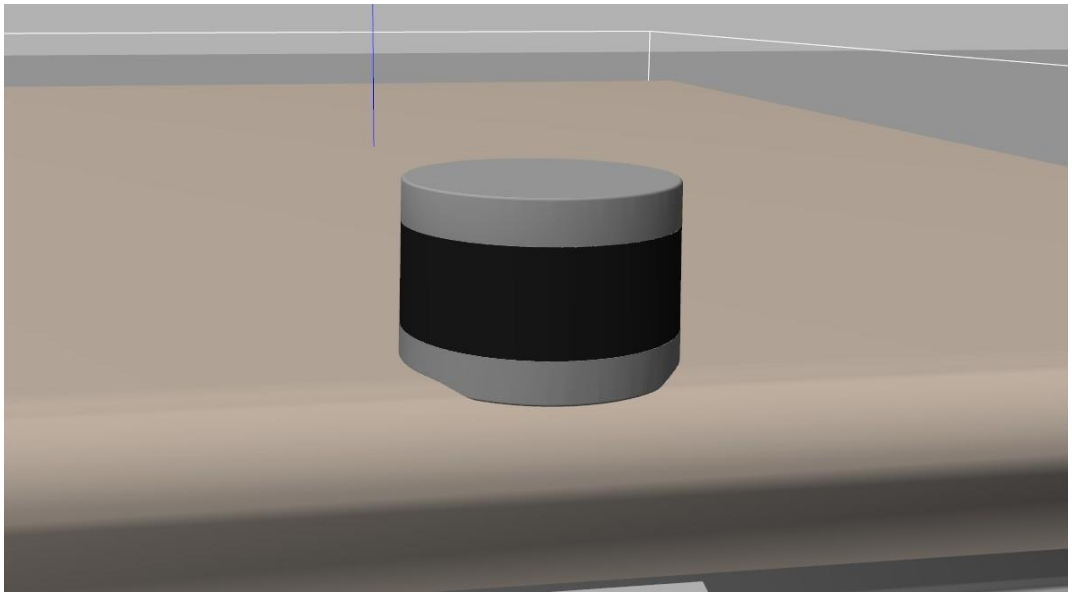


Ilustración 24 Velodyne

Para facilitar la lectura del código, tenerlo más ordenado y para que sea más ágil a futuros cambios se ha empaquetado todo el código del sensor en un formato MACRO dentro del archivo VLP-16.urdf.xacro, archivo que se incluye en el fichero rbcar.urfd.xacro, que es el encargado de construir el modelo en Gazebo.

```
<VLP-16 parent="base_link" name="velodyne" topic="/velodyne_points">
  <origin xyz="0.62 0 1.75" rpy="0 0 0" />
</VLP-16>
```

Esta MACRO tiene cuatro atributos para poder configurar el sensor.

parent: nombre del eslabón al que se va a añadir el sensor.

name: nombre del eslabón del sensor.

topic: nombre del topic que publica la nube de puntos.

origin: posición y orientación que ocupará el sensor.

6.5 ADICIÓN DE LA CÁMARA ESTÉREO

ROS ofrece una gran cantidad de plugins que permiten ampliar el rango de funcionalidades que puede realizar el robot. Una de ellas es la posibilidad de añadir una interfaz para simular cámaras, sean convencionales o incluso múltiples, que permiten ser visualizadas en tiempo real en RViz.

Para representar las cámaras de manera visual, se ha creado una caja en forma de cubo que se ha añadido al robot. Sobre este nuevo elemento se han configurado todos los parámetros que permiten la simulación de la cámara.

```
<link name="${name}">
      <visual>
        <geometry>
           <box size= "0.1 0.1 0.1"/>
        </geometry>
      </visual>

     <collision>
        <geometry>
           <box size= "0.1 0.1 0.1"/>
        </geometry>
      </collision>

</link>
<joint name="base_link_to_${name}" type="fixed">
      <xacro:insert_block name="origin" />
      <parent link="${parent}"/>
      <child link="${name}"/>
</joint>
```

De este modo ya se ha añadido el eslabón que va a contener la cámara al robot, a falta de configurarla.

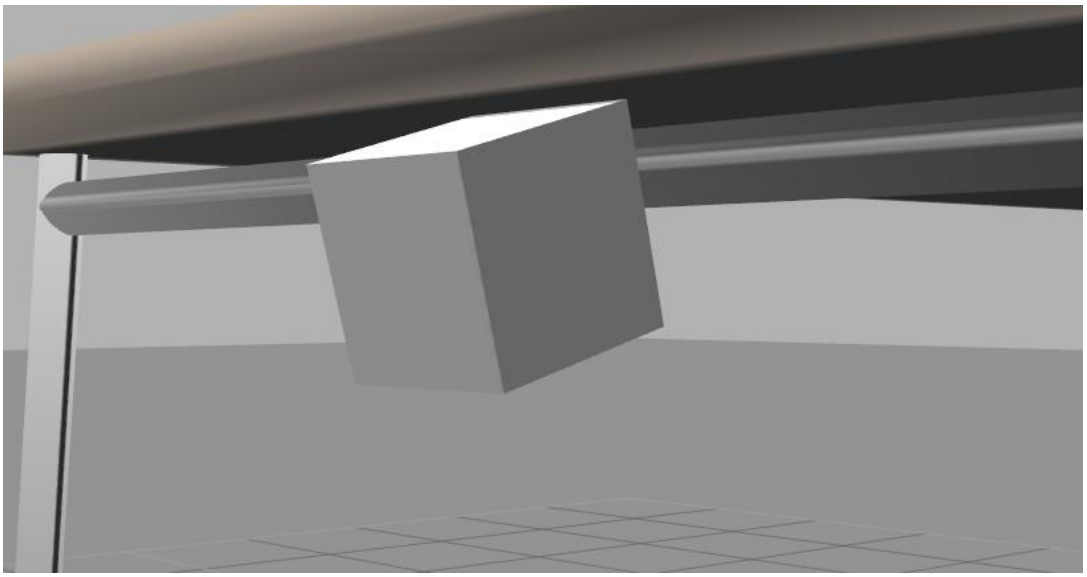


Ilustración 25 Posición de la cámara en el modelo

Para añadir el plugin, primero se define un nombre para el sensor, después se selecciona el tipo de sensor que se va a configurar y el nombre del eslabón al que se va a añadir. Posteriormente se configura la frecuencia de actualización en Hz, en este caso se ha optado por 10 Hz. Esto se puede ver a continuación:

```
<gazebo reference="${name}">
      <sensor type="multicamera" name="stereo_camera">
      <update_rate>10.0</update_rate>
```

A continuación, se procede a configurar cada una de las cámaras de manera separada, tanto la izquierda como la derecha.

```
<camera name="left">
         <pose>0 -0.12 0 0 0 0</pose>
        <horizontal_fov>0.750492</horizontal_fov>
        <image>
          <width>1280</width>
          <height>960</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
        <noise>
```

```
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
 </camera>
```

Primero se establece la posición de la cámara respecto a la posición del eslabón, es decir, al ser la izquierda se desplazada ligeramente hacia ese sentido. Después se configura el tipo y el tamaño de imagen que va a capturar el sensor. Finalmente, se fija la distorsión de la cámara, que en ese caso es nula.

Respecto a la cámara derecha, se ha configurado de forma exactamente igual, salvo la posición de la cámara, que en este caso se mueve ligeramente la misma distancia hacia la derecha respecto del punto de referencia en el eslabón.

```
<camera name="right">
        <pose>0 0.12 0 0 0 0</pose>
```

Finalizando la configuración de la cámara, se define el pulgin que la va a controlar y que va a permitir su visualización en RViz, donde se define el nombre del topic que va a publicar. Los demás valores se dejan igual a la configuración por defecto [18].

```
      <plugin name="stereo_camera_controller"
         filename="libgazebo_ros_multicamera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>stereo_camera/</cameraName>
        <imageTopicName>image_rect_color</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>stereo_camera</frameName>
       <hackBaseline>0.12</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
      </plugin>
    </sensor>
</gazebo>
```

Al igual que se ha hecho con otras partes del robot, se ha insertado el contenido del sensor dentro de una MACRO en el archivo stereo_CAM.urdf.xacro. En el archivo donde se fij el conjunto del robot se define de la siguiente forma:

```
<Stereo_CAM parent="base_link" name="stereo_camera">
   <origin xyz="0.70 0 1.62" rpy="0 0.2617 0"/>
</Stereo_CAM>
```

Donde presenta tres atributos:

parent: nombre del eslabón al que se va a añadir el sensor.

name: nombre del eslabón del sensor.

origin: posición y orientación que ocupará el sensor.

6.6 ADICIÓN DEL SENSOR LASER

Otra de las posibilidades que se ofrece en ROS-Gazebo en cuanto a sensores y plugins es la inserción de sensores laser y posteriormente visualizar los datos capturados en RViz.

En este caso, para representar físicamente el laser, se crea un cubo de tamaño muy reducido que se va a añadir en la parte delantera inferior del vehículo.

```
<link name="${name}">
      <visual>
        <pose>0 0 0 0 0 0</pose>
        <geometry>
           <box size= "0.01 0.01 0.01"/>
        </geometry>
      </visual>

     <collision>
        <pose>0 0 -0 0 0 0</pose>
        <geometry>
           <box size= "0.01 0.01 0.01"/>
        </geometry>
      </collision>
</link>

<joint name="base_link_to_${name}" type="fixed">
    <xacro:insert_block name="origin" />
    <parent link="${parent}"/>
```

```
    <child link="${name}"/>
</joint>
```

A continuación, se configuran las características que tendrá el sensor, primero se fija su nombre, su posición y el tipo de sensor y la frecuencia de actualización en Hz.

```
<gazebo reference="${name}">
  <sensor type ="gpu_ray" name="hoyuko">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>20</update_rate>
```

Seguidamente, los parámetros del rendimiento que ofrece el sensor. El número de muestras que se toman y la resolución de las mismas, además de los ángulos máximos de captura del sensor a izquierda y a derecha, ambos en radianes.

```
<ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.0472</min_angle>
          <max_angle>1.0472</max_angle>
        </horizontal>
      </scan>
```

Posteriormente se fija el alcance del laser, el mínimo y el máximo, en metros.

```
      <range>
        <min>0.10</min>
        <max>2.5</max>
        <resolution>0.01</resolution>
      </range>
</ray>
```

El rango de acción del será casi semicircular en la parte delantera del robot, no se ha definido la semicircunferencia completa para evitar la detección constante de las ruedas delanteras.

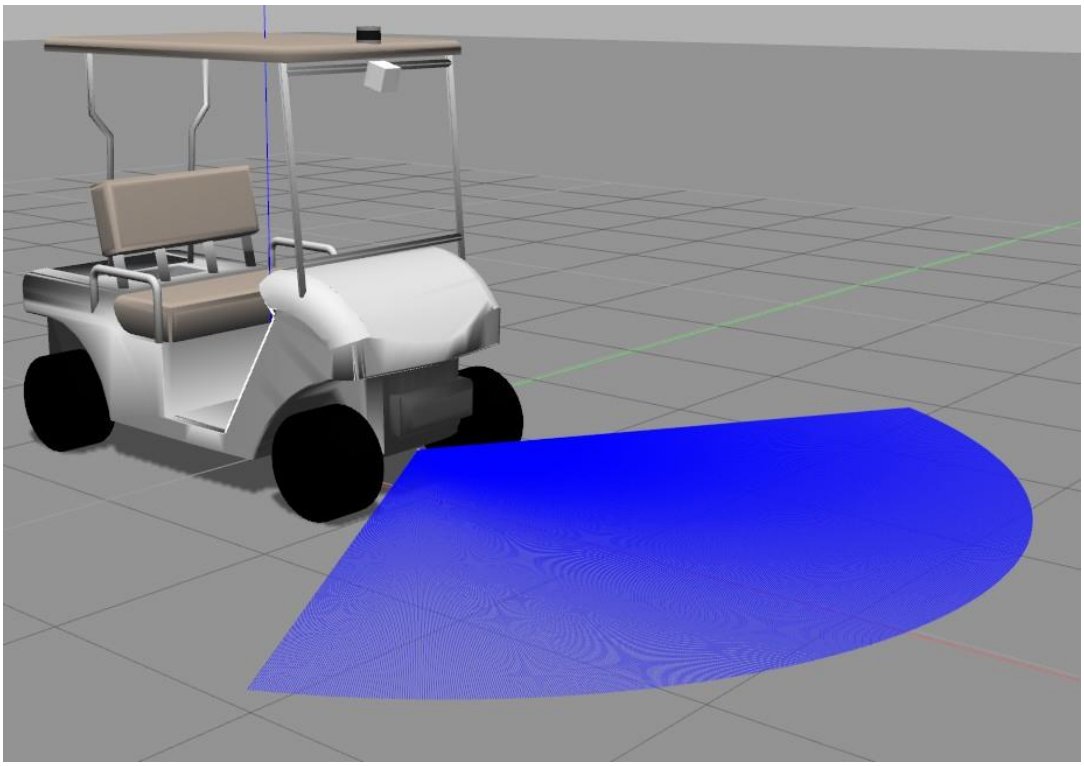


Ilustración 26 Rango de acción del laser

Finalmente, se configura el plugin para su visualización en RViz, donde se establece el nombre del topic que se publicará.

```
    <plugin name="gpu_laser" filename="libgazebo_ros_gpu_laser.so">
      <topicName>${topic}</topicName>
      <frameName>${name}</frameName>
    </plugin>
  </sensor>
</gazebo>
```

Del mismo modo que los demás sensores del vehículo, el laser se ha incluido en una MACRO dentro del archivo LASER.urdf.xacro, después en el archivo, rbcar.urdf.xacro se llama con la siguiente sentencia:

```
  <LASER parent="base_link" name="laser" topic="/scan">
    <origin xyz="0.92 0 0.04" rpy="0 0 0" />
  </LASER>
```

Con los siguientes atributos:

parent: nombre del eslabón al que se va a añadir el sensor.

name: nombre del eslabón del sensor.

topic: nombre del topic que publica la nube de puntos.

origin: posición y orientación que ocupará el sensor

6.7 CREACIÓN DE UN ENTORNO DE PRUEBAS

6.7.1 Creación del mapa:

Se necesita un entorno donde realizar pruebas para poder evaluar el vehículo que se ha modelado. Para ello, se parte de una imagen .png que va a servir como referencia a la hora de diseñar este espacio. A partir de esta imagen se va a crear un mapa en RViz y un mundo de Gazebo.

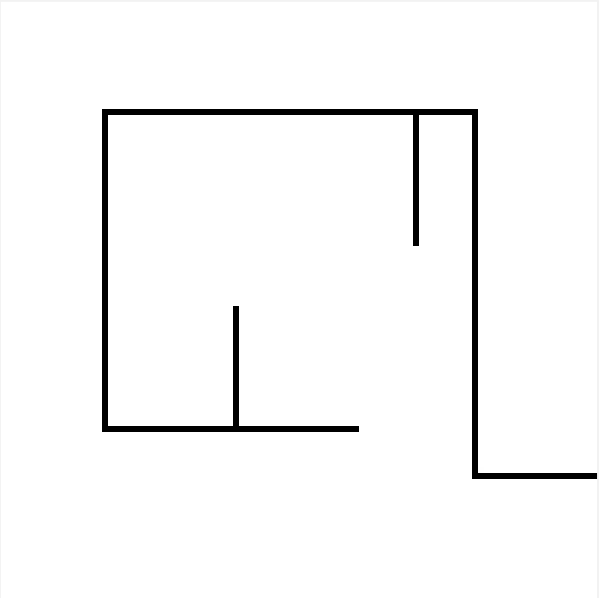


Ilustración 27 Imagen que se utiliza para crear el mapa

El mapa en RViz, se define y publica en un archivo .launch, tf.launch, donde se publican dos nodos de ROS.

```
<launch>

    <node pkg="tf" type="static_transform_publisher" name="initial_loc"
        args="25 25 0 0 0 0 /map /odom 2" output="screen"/>


    <!-- global map, sShould be outside of the namespace -->

    <node name="map_server" pkg="map_server" type="map_server"

        args="$(find rbcar_gazebo)/maps/map1.yaml" output="screen">

       <param name="frame_id" value="map"/>

    </node>
```

```
</launch>
```

El primero de ellos, publica las transformadas entre el origen del mapa y el nodo odom, que está situado en la posición inicial del vehículo. El origen de coordenadas está situado en la esquina inferior de la imagen, así pues, como se quiere tener el vehículo situado en el centro, se establece en las coordenadas x e y el valor de 25.

El otro publica el mapa a partir del nodo tipo map_server, que permite ofrecer un mapa como un servicio de ROS. Los parámetros están definidos en el archivo map1.yaml

```
image: map1.png
resolution: 0.5
origin: [0.0, 0.0, 0.0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Se establece la imagen que servirá de referencia, map1.png. También se ajusta la resolución del mapa, en metros/pixel, al ser una imagen de 100x100px se establece una resolución de 0,5. Por último, se puede configurar la orientación del mapa en el campo origin, tomando como referencia la esquina inferior izquierda de la imagen. En este caso no se ha editado, y se han dejado las tres orientaciones en 0. Los demás campos, occupied_thresh, free_thresh y negate se han dejado con sus valores por defecto, que vienen definidos en la documentación de ROS [19].

Una vez configurado el mapa, se puede visualizar en RViz en el topic map.

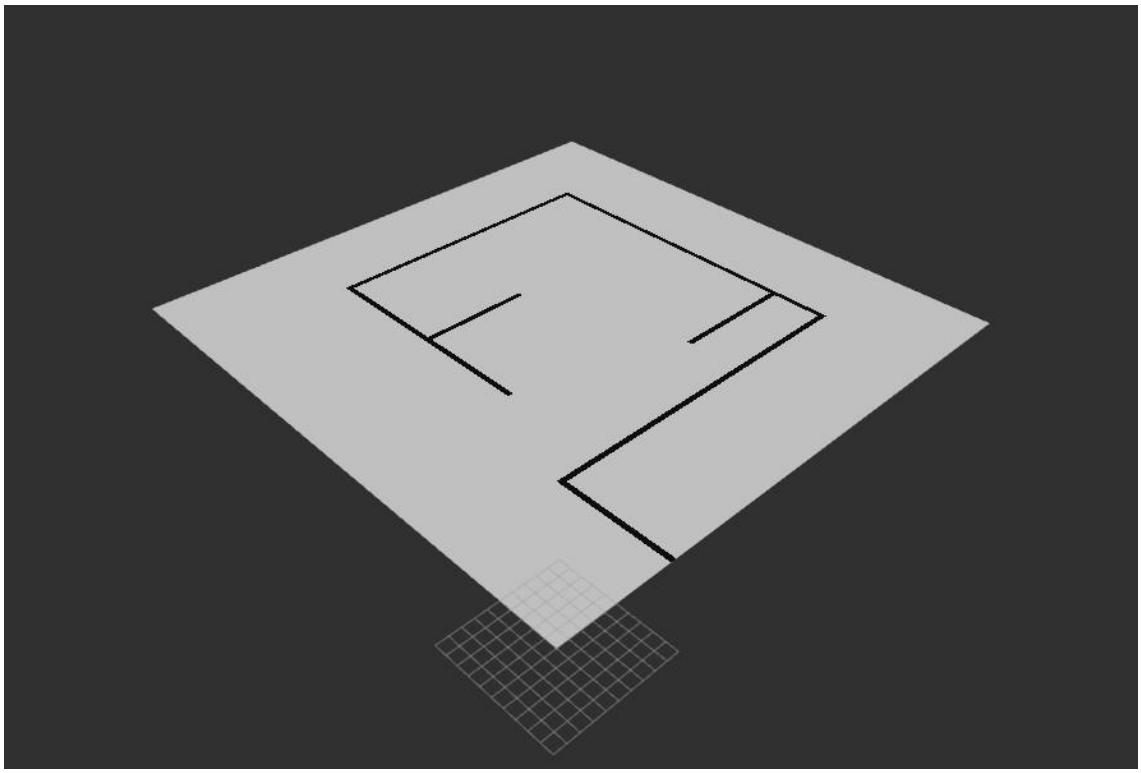


Ilustración 28 Mapa en Rviz

6.7.2 Creación del mundo en Gazebo:

Una de las múltiples opciones que permite Gazebo es crear mundos, archivos .world, con la herramienta, *Building Editor*, donde se pueden extruir una imagen en dos dimensiones y crear un espacio tridimensional.

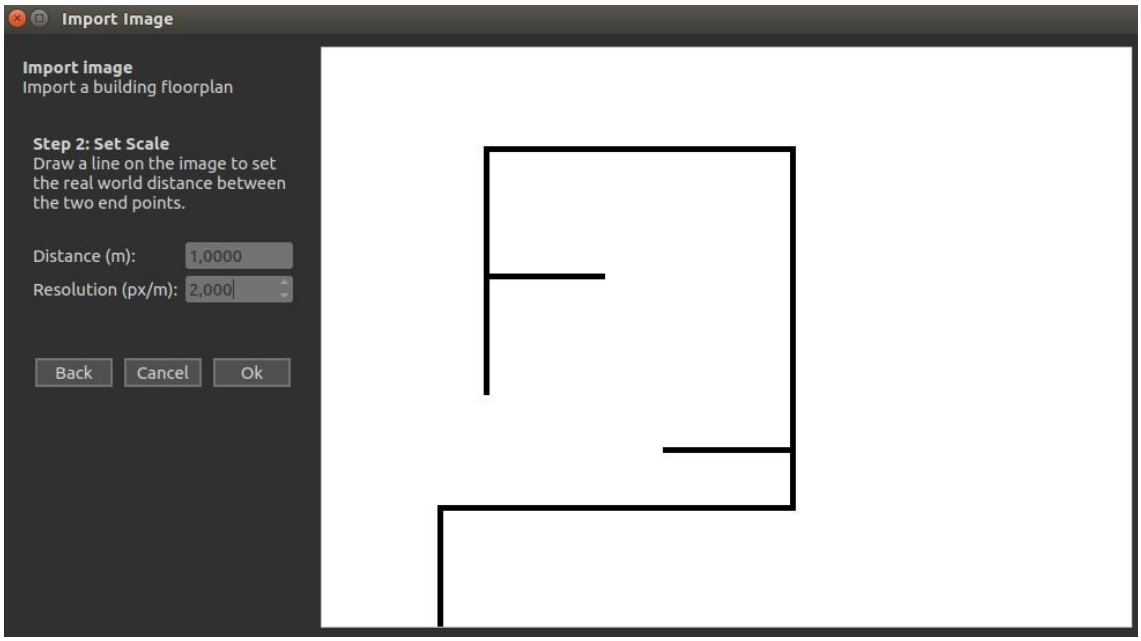


Ilustración 29 Gazebo Building Editor

En el apartado anterior, sobre esta misma imagen se ha creado un mapa con una resolución de 0,5 metros/pixel. De este modo, para que el tamaño del mundo sea igual, la resolución en este caso deberá ser 2 pixeles/metro.

El mundo se guarda con el nombre de map2_s.

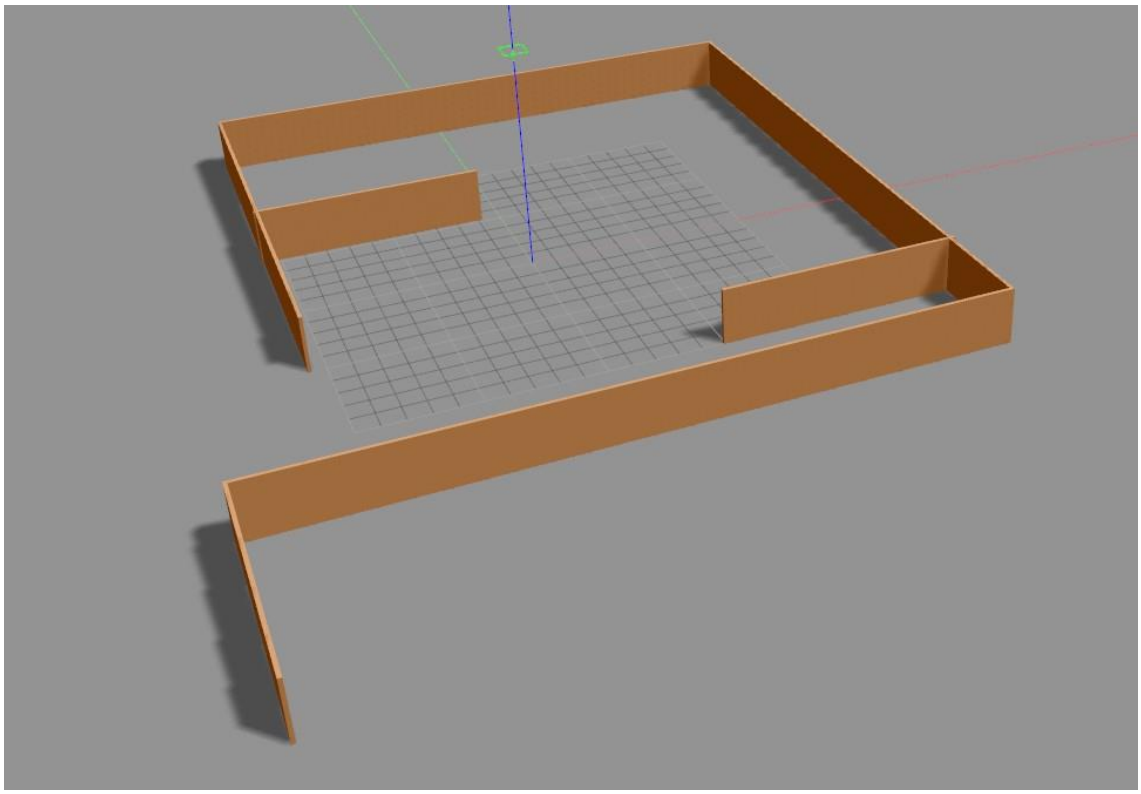


Ilustración 30 Mundo de simulación

Una vez, creado el modelo, se debe de incluir en el archivo que publica el mundo completo en la simulación, en este caso, rbcar.world.

```
<sdf version="1.4">
  <world name="default">
    <physics type="ode">
      <gravity>0 0 -9.8</gravity>
      <ode>
        <solver>
          <!--type>quick</type>
          <dt>0.001</dt>
          <iters>40</iters>
          <sor>1.0</sor -->
          <!-- type>quick</type>
          <dt>0.01</dt>
          <iters>20</iters>
```

```
          <sor>1.0</sor -->
          <type>quick</type>
          <!--dt>0.001</dt-->
          <iters>20</iters>
          <sor>1.0</sor>
        </solver>
        <constraints>
          <cfm>0.0</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>100.0</contact_max_correcting_vel>
          <contact_surface_layer>0.0</contact_surface_layer>
        </constraints>
      </ode>
      <max_step_size>0.001</max_step_size>
    </physics>
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://map2_s</uri>
    </include>
  </world>
</sdf>
```

En este archivo primero se definen las físicas del mundo que se va a simular, donde se han dejado los valores por defecto.

```
<physics type="ode">
      <gravity>0 0 -9.8</gravity>
      <ode>
        <solver>
          <!--type>quick</type>
          <dt>0.001</dt>
```

```
        <iters>40</iters>
        <sor>1.0</sor -->
        <!-- type>quick</type>
        <dt>0.01</dt>
        <iters>20</iters>
        <sor>1.0</sor -->
        <type>quick</type>
        <!--dt>0.001</dt-->
        <iters>20</iters>
        <sor>1.0</sor>
      </solver>
      <constraints>
        <cfm>0.0</cfm>
        <erp>0.2</erp>
        <contact_max_correcting_vel>100.0</contact_max_correcting_vel>
        <contact_surface_layer>0.0</contact_surface_layer>
      </constraints>
    </ode>
    <max_step_size>0.001</max_step_size>
  </physics>
```

A continuación, se definen todos los modelos que se van a incluir en la simulación, además de los modelos ground_plane y sun, que vienen por defecto, se añade el modelo creado map2_s.

```
  <include>
    <uri>model://sun</uri>
  </include>
  <!-- A ground plane -->
  <include>
    <uri>model://ground_plane</uri>
  </include>
  <include>
    <uri>model://map2_s</uri>
  </include>
```

6.8 LOCALIZACIÓN DEL VEHÍCULO

Para poder controlar en cualquier instante la posición del modelo, se ha utilizado el paquete gazebo_odometry. Este paquete permite obtener la posición y orientación del vehículo.

Se entiendo como odometría el estudio de la posición de vehículos con ruedas de navegación [20]. Por lo que resulta fundamental controlar la odometría para poder localizar el modelo en las simulaciones.

La odometría se publica a partir del archivo .launch, gazebo_odometry.launch

```
<launch>

        <param name="model_subs" value="gazebo/model_states"/>

        <param name="model_name" value="rbcar"/>

        <param name="odom_topic" value="gazebo_odometry"/>


        <node pkg="gazebo_odometry" type="gazebo_odometry"
        name="gazebo_odometry" output="screen"/>

</launch>
```

En este archivo, primero se define el modelo del cual se pretende publicar la odometría, en este caso el rbcar. Sobre este modelo, concretamente sobre su eslabón base_footprint, se va a publicar la odometría. De este modo, a partir de las transformadas de este eslabón, se podrá obtener la posición relativa de cada una de las partes del modelo completo.

Después de haber definido los parámetros, se lanza el nodo gazebo_odometry. Este nodo va a ser el encargado de publicar odom, que va a permitir conectar el mapa con el vehículo, manteniendo el mapa como punto fijo de referencia en el espacio.

De este modo, la estructura de todo la odometría del modelo es la siguiente:

Ilustración 31 Odometría completa

6.9 SIMULACIÓN DEL MODELO COMPLETO

Para poder ejecutar todas las partes del modelo y además poder controlarlas, se ha establecido un archivo .launch, rbcar.launch. En este archivo, se lanzan todas las partes que se han ajustado en apartados anteriores.

```
<launch>


  <include file="$(find rbcar_control)/launch/rbcar_control.launch" />

  <include file="$(find
        rbcar_robot_control)/launch/rbcar_robot_control.launch" />


  <include file="$(find gazebo_odometry)/launch/gazebo_odometry.launch" />

  <include file="$(find rbcar_gazebo)/launch/tf.launch" />


  <include file="$(find gazebo_ros)/launch/empty_world.launch">

    <arg name="world_name" value="$(find rbcar_gazebo)/worlds/rbcar.world"/>

    <arg name="paused" value="false"/>

    <arg name="use_sim_time" value="true"/>

    <arg name="headless" value="false"/>

  </include>


  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
        rbcar_description)/robots/rbcar.urdf.xacro'" />


  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        respawn="false" output="screen" args="-urdf -model rbcar -param
        robot_description -z -0.04"/>


</launch>
```

Primero se lanzan dos ficheros .launch, el primero sirve para inicializar el controlador del coche y el segundo su función es publicar todos los topics que permiten el control del vehículo. Se ejecuta el topic /rbcar_robot_control/command, sobre el cual se publican mensajas del tipo /ackermann_msgs, con los que se podrá controlar el movimiento del coche.

Ambos archivos son propios del paquete por defecto, no se han modificado, salvo que se han incluido en este archivo para poder realizar la simulación para lanzar todos los nodos de ROS de manera simultánea.

```
  <include file="$(find rbcar_control)/launch/rbcar_control.launch" />

  <include file="$(find
        rbcar_robot_control)/launch/rbcar_robot_control.launch" />
```

A continuación, se lanzan los archivos que publican la odometría y las transformadas respecto al mapa, se han visto en los apartados .8 y .7.1 respectivamente.

```
<include file="$(find gazebo_odometry)/launch/gazebo_odometry.launch" />

<include file="$(find rbcar_gazebo)/launch/tf.launch" />
```

Seguidamente, se incluye el fichero encargado de publicar el mundo de Gazebo, además se definen varios parámetros, como si se empieza la simulación pausada o si se va a contar el tiempo de simulación.

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">

  <arg name="world_name" value="$(find rbcar_gazebo)/worlds/rbcar.world"/>

  <arg name="paused" value="false"/>

  <arg name="use_sim_time" value="true"/>

</include>
```

Finalmente, se establece como parámetro el modelo .urdf.xacro que se va a usar como robot y posteriormente se utiliza este parámetro para ejecutar el nodo que permite que aparezca este modelo en Gazebo. Para evitar que el modelo se lance al aire, se introduce un pequeño offset negativo, de este modo, el modelo aparecerá directamente en el suelo.

```
<param name="robot_description" command="$(find xacro)/xacro.py
      '$(find  rbcar_description)/robots/rbcar.urdf.xacro'" />


<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
      respawn="false" output="screen" args="-urdf -model rbcar -param
      robot_description -z -0.04"/>
```

7 RESULTADOS

Una vez creado el modelo y el entorno de pruebas, se realizan una serie de simulaciones para poder evaluar los desarrollos y el cumplimiento de los objetivos establecidos en el apartado 1.2.

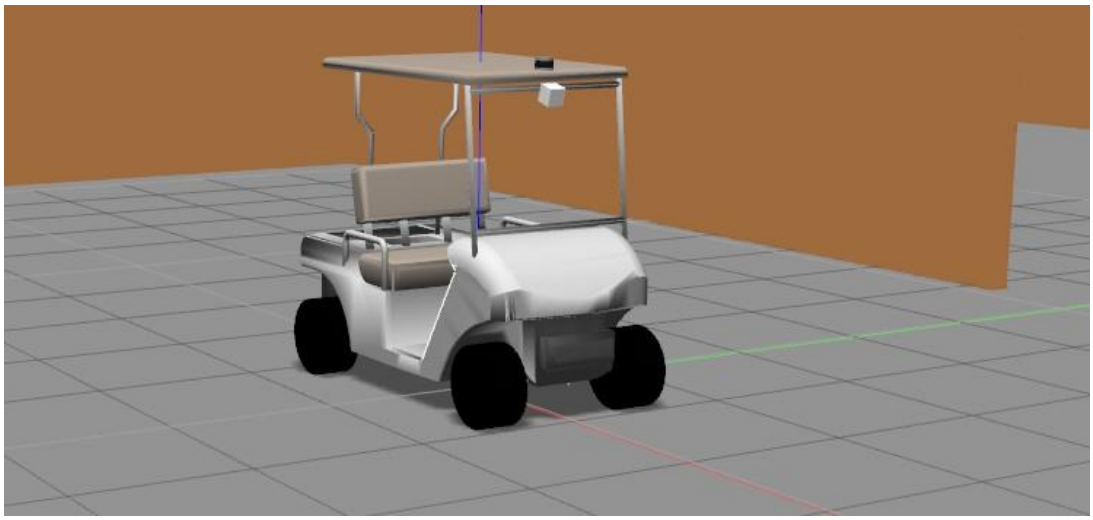


Ilustración 32 Modelo final completo

Se va a evaluar el correcto funcionamiento del sistema de dirección, del sistema de colisiones y de la todos los sistemas de percepción.

7.1 SISTEMA DE DIRECCIÓN

El sistema de dirección debe cumplir la geometría de Ackermann, es decir, la posición de las ruedas delanteras al realizar los giros debe ajustarse a los requisitos definidos en el apartado 3.

Se observa las ruedas en su estado inicial:

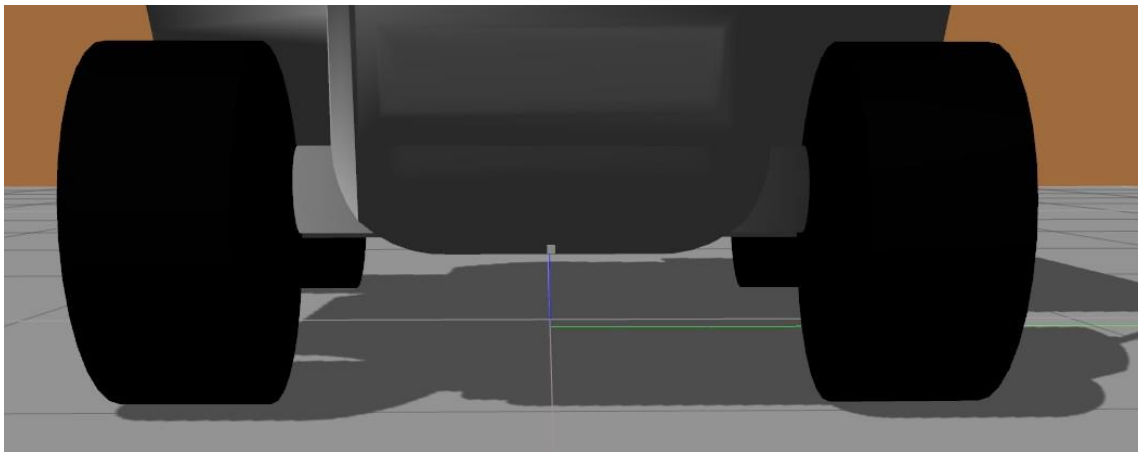


Ilustración 33 Ruedas delanteras en posición inicial

Las ruedas realizando un giro a la derecha:

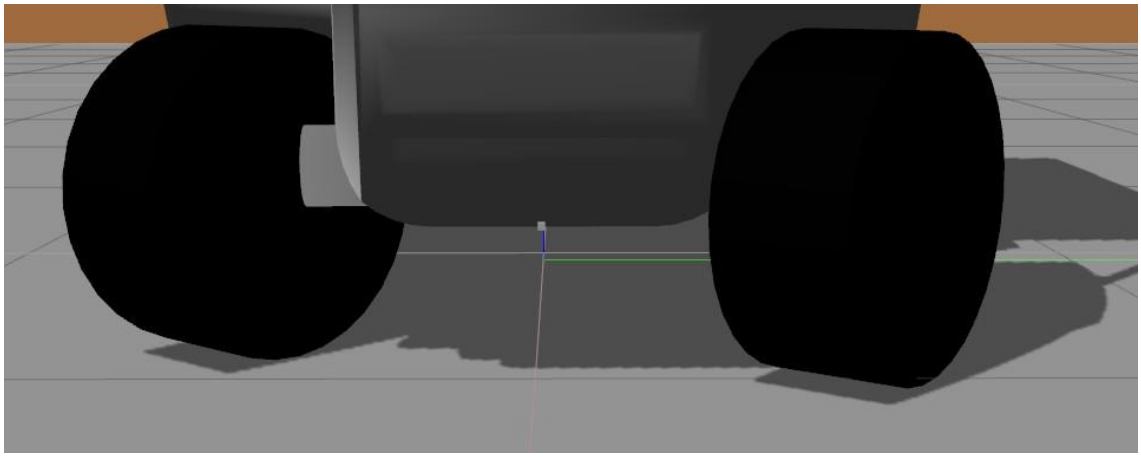


Ilustración 34 Ruedas delanteras realizando giro a derecha

Y finalmente, las ruedas delanteras realizando un giro a la izquierda:

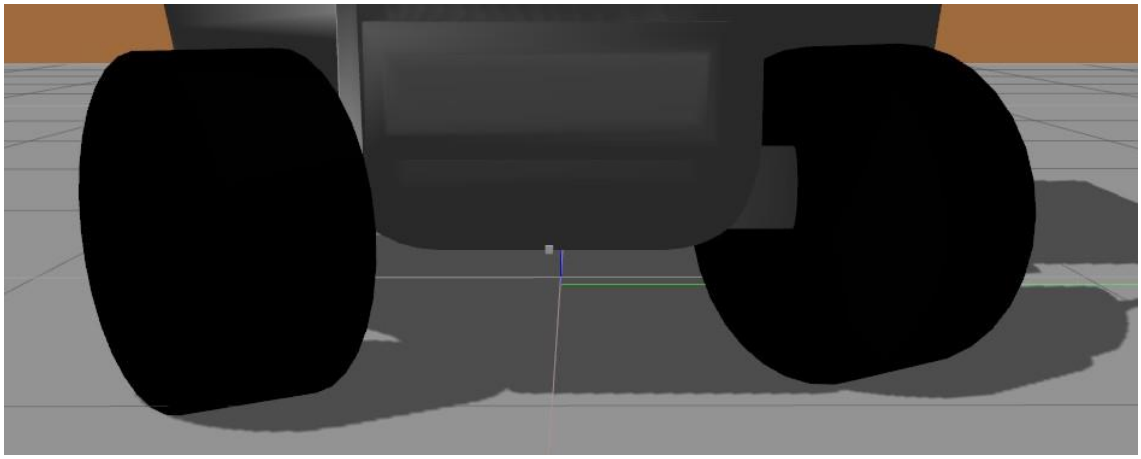


Ilustración 35 Ruedas delanteras realizando giro a izquierda

Por otro lado, se puede observar las ruedas traseras, que la distancia entre ellas debe de ser mayor que en las ruedas delanteras:

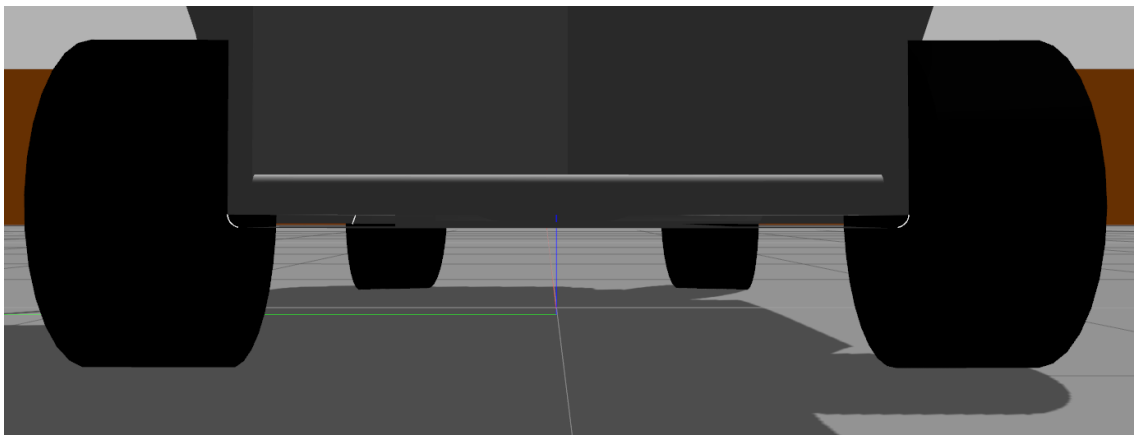


Ilustración 36 Ruedas traseras

7.2 SISTEMA DE COLISIÓN

Para comprobar que el modelo de colisiones del vehículo se ha diseñado correctamente, se mueve el coche hasta forzar el choque del modelo con las paredes del entorno de pruebas.

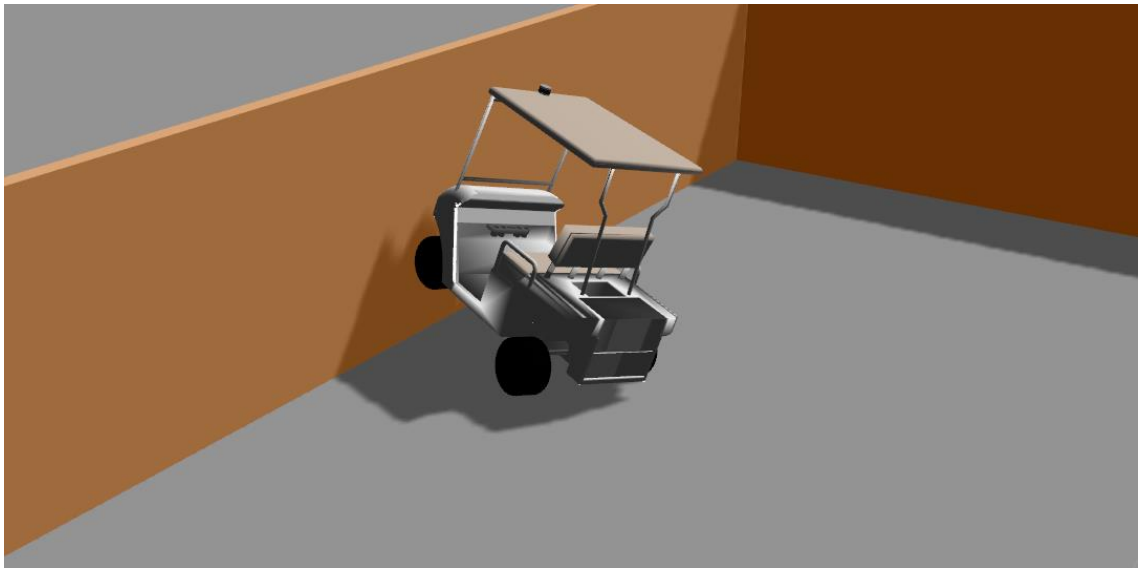


Ilustración 37 Colisión del vehículo

En el momento que el vehículo alcanza la pared y los motores siguen impulsándolo, el coche choca con el muro y se levanta ligeramente.

7.3 SISTEMAS DE PERCEPCIÓN

7.3.1 Sensor Velodyne:

El sensor Velodyne debe de ser capaz de detectar todos los elementos que rodeen al vehículo y representarlos mediante nubes de puntos. Para poder evaluar su comportamiento en el entorno de simulación se ha situado el vehículo en el centro, lugar desde donde debe detectar las paredes del mapa que están situadas a distintas distancias. Los resultados se pueden visualizar en Rviz, mediante el topic PointCloud2.

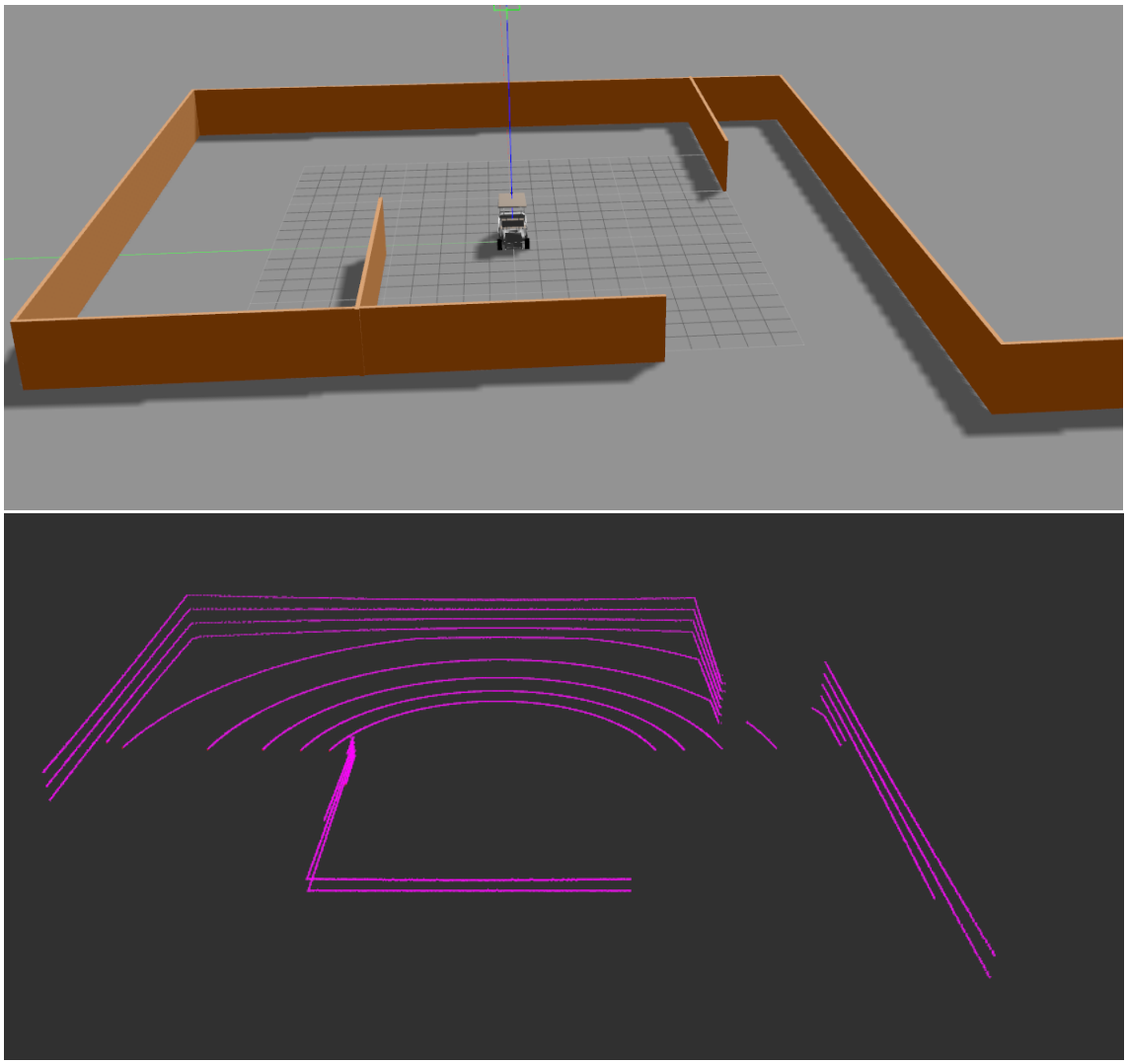


Ilustración 38 Detección del velodyne en Rviz

En la imagen superior se puede comprobar todas las paredes es capaz de detectar el sensor. En la siguiente imagen se puede apreciar en Rviz la detección en tres dimensiones, se observa cómo se captura la altura de los muros, primero en relación al mapa del entorno y después en un escenario vacío:

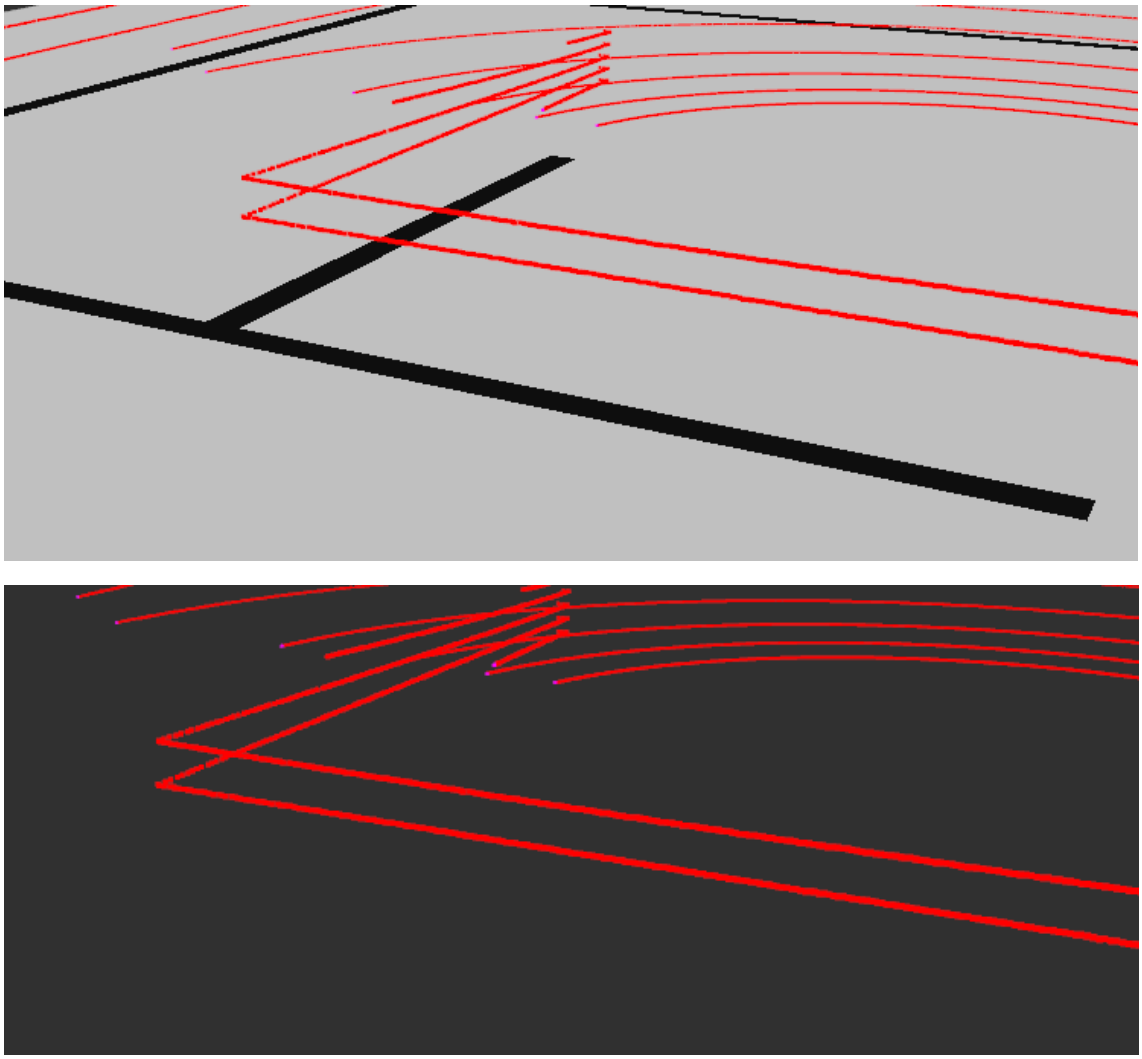


Ilustración 39 Detección de una pared en Rviz

7.3.2 Cámara estéreo:

La cámara del vehículo permite observar en tiempo real todo los elementos que se sitúen en frente del vehículo. Hay dos cámaras separadas entre ellas 20 cm, se pueden ver el contenido que están capturando mediante RViz.

Para evaluar el funcionamiento de las cámaras se ha movido el coche hacia una posición donde se pueda apreciar la diferencia de perspectiva entre las dos cámaras.

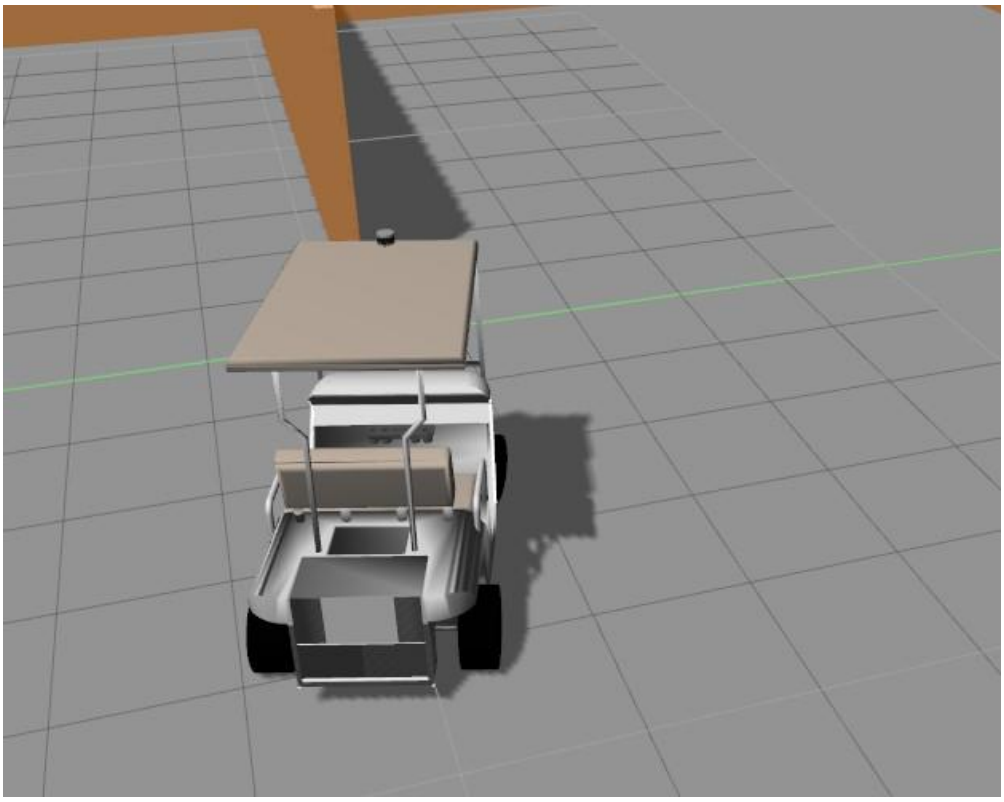


Ilustración 40 Posición de evaluación de las cámaras

Desde esta posición el contenido que se observa es el siguiente:

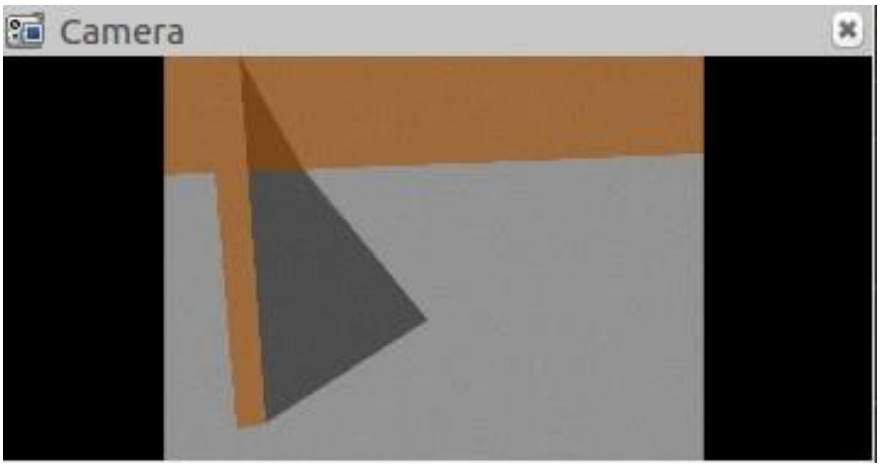


Ilustración 41 Captura de la cámara derecha

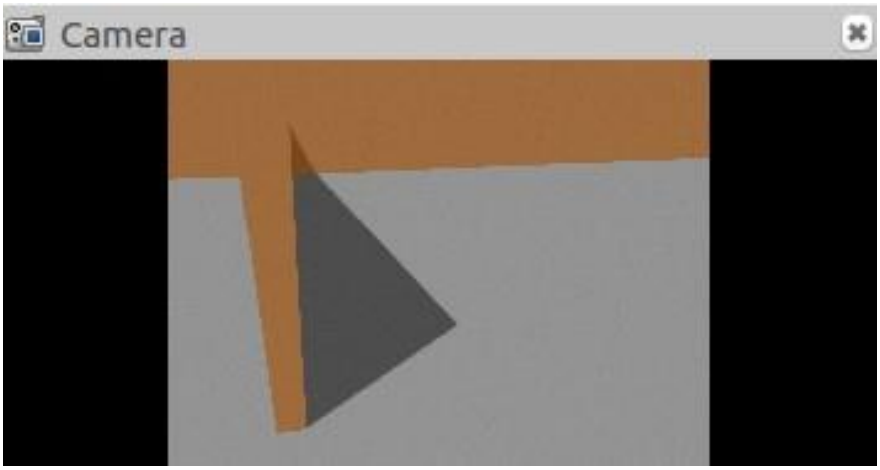


Ilustración 42 Captura de la cámara izquierda

En la imagen capturada desde la derecha, se puede apreciar que la parte sombreada es mayor debido a su perspectiva. En la cámara de la izquierda, la se aprecia más claramente la parte exterior de la pared.

7.3.3 Sensor laser:

El sensor laser detecta los elementos que están en la parte inferior delantera del vehículo. De este modo, para evaluar su comportamiento, se mueve el vehículo hasta una posición que permita detectar los elementos y que puedan ser visualizados en RViz.

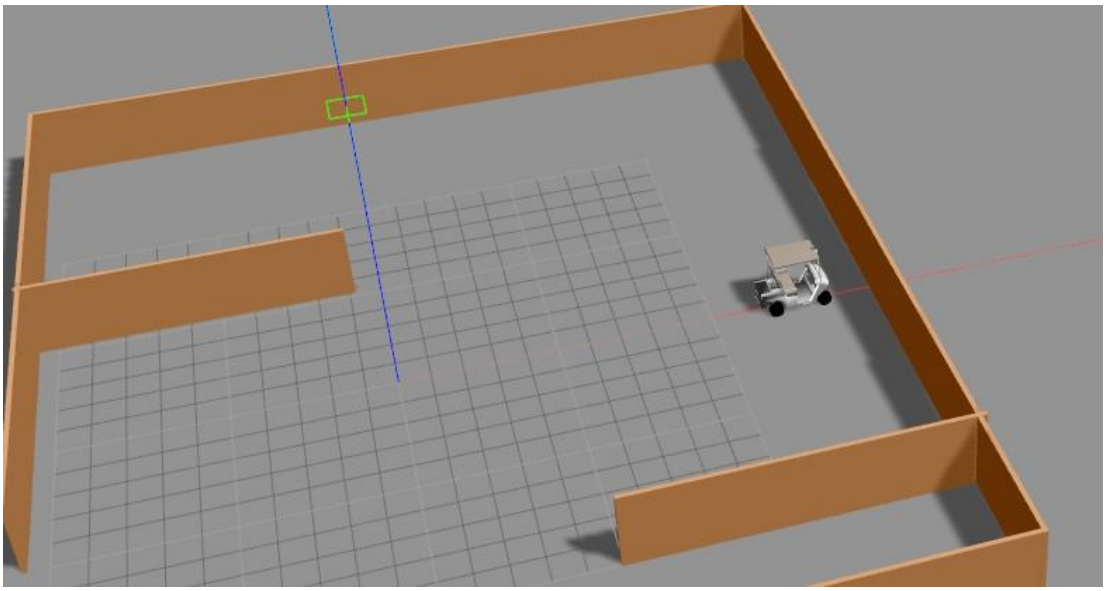


Ilustración 43 Posición de evaluación del laser

Desde esta posición, en RViz el resultado de la detección es el siguiente, comparado con el mapa:

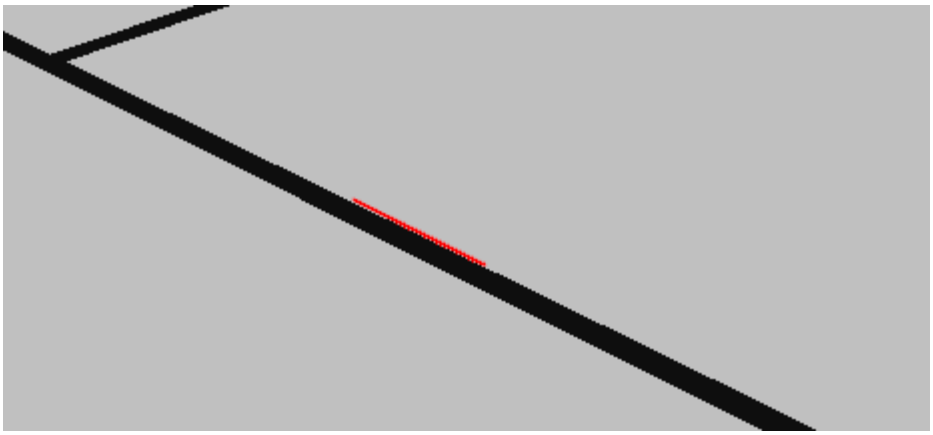


Ilustración 44 Detección del laser en el mapa

Se puede apreciar una ligera línea roja delante de la pared, para poder apreciarla de manera más clara, sin el mapa la detección es la siguiente:

Ilustración 45 Detección del laser

El laser tiene un rango de detección semicircular, para poder evaluarlo se han colocado una serie de elementos de prueba delante del coche.

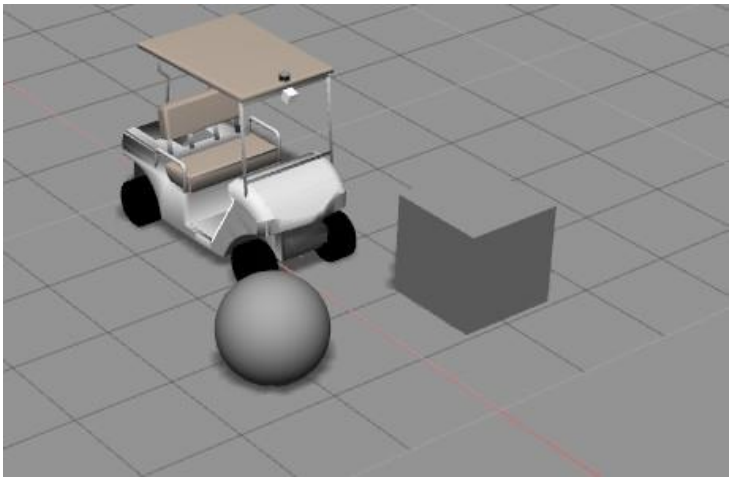


Ilustración 46 Elementos para evaluar el laser

La observación en RViz es la siguiente, vista en planta:

Ilustración 47 Observación de los elementos de test

8 CONCLUSIÓN

El objetivo de este proyecto era desarrollar y modelizar un modelo que se ajustara al modelo real y posteriormente realizar simulaciones para comprobar si el comportamiento del vehículo simulado se ajusta a la realidad.

Tras haber llevado a cabo todos los desarrollos, se concluye que los objetivos principales se han cumplido de manera satisfactoria. Se ha conseguido crear el modelo tridimensional sobre el cual se ha implementado la geometría de Ackermann. Asimismo, también se han logrado implementar todos los sistemas de percepción sobre el modelo simulado. Al realizar distintas simulaciones se ha podido comprobar que se ha conseguido que el comportamiento de cada uno de los sensores y de la cámara se ajusta a los objetivos que se habían establecido al principio de este trabajo.

En cuanto se refiere al entorno de simulación, se ha conseguido crear un entorno donde se han podido realizar las distintas pruebas. No obstante, para que se pueda reutilizar el vehículo que se ha modelado en otros entornos de Gazebo se deberá de reajustar la publicación de la odometría de cada una de las partes del modelo. De no ser así, no se podrá controlar con exactitud la posición en tiempo real del vehículo en el mapa.

Se debe de tener en cuenta que, para poder ejecutar el conjunto completo, el modelo con todos los sensores en el mapa simulado se requiere de una gran capacidad en la máquina donde tiene lugar el proceso. Como punto de mejora se puede optimizar la capacidad computacional que requieran las simulaciones o llevarlas a cabo en un ordenador con los recursos suficientes.

Para futuros desarrollos, sobre el modelo simulado, se podría optimizar el comportamiento del controlador del vehículo, sobre todo en cuanto al sistema de dirección se refiere. A partir de un cierto ángulo de giro las ruedas quedan en una posición irreversible y el modelo pasa a ser incontrolable. Del mismo modo, para optimizar el controlador de dirección, se podrían calcular e incorporar los parámetros de controlador PID que incorpora, donde se han dejado los valores que venían por defecto.

Otro futurible desarrollo sería la creación de un entorno virtual que simule el campus de la UC3M de Leganés, ya que este vehículo se ha construido para moverse en este espacio en la realidad.

9 PLANIFICACIÓN Y PRESUPUESTO

9.1 PLANIFICACIÓN

Para poder lograr los objetivos establecidos, se ha dividido el proyecto en tres fases, cada una de ellas imprescindible:

Fase 1: Definición del problema y recopilación de información (2 semanas)

En estas dos primeras semanas se define la necesidad y se exploran posibles soluciones para poder llevar a cabo el desarrollo más óptimo.

Fase 2: Aprendizaje y familiarización de los componentes de ROS (4 semanas)

Se realizan los tutoriales de ROS que están en la red a fin de aprender y comprender los conceptos básicos de este entorno.

Fase 3: Ejecución y desarrollo (20 semanas)

En este periodo se desarrolla la modelización del proyecto, siguiendo los pasos que se han detallado en el capítulo 6 de esta memoria. Posteriormente, se llevan a cabo las simulaciones y la recopilación de resultados En esta fase también se incluye la redacción de la presente documentación.

9.2 PRESUPUESTO

En este proyecto se han utilizado la mayoría de herramientas gratuitas de código abierto. No obstante, para calcular el presupuesto total también se deben de tener en cuenta el ordenador y el sueldo del desarrollador.

El ordenador tendría un coste de 899€, teniendo en cuenta que tiene una vida útil de 4 años y se ha utilizado durante 26 semanas, equivalente a medio año, se ha gastado una octava parte de la vida útil de la máquina, lo que tiene un valor de 112,35€.

Por otro lado, las horas de trabajo. 1 crédito ECTS equivale 25 horas de trabajo, a este proyecto se le han asignado 12 ECTS, lo que resultan 300 horas. El sueldo de un becario se establece a 6,25€ cada hora, por lo que en total equivaldría 1875€.

De este modo el presupuesto total es el siguiente:

Ítem	Descripción del elemento	Cantidad	Precio unitario	Precio total
1	Portátil HP 15-bs027ns, i5, 16 GB, 1 TB, Radeon 530 4 GB [21]	1	112,35€	112,35€
2	SO Ubuntu 16.04	1	0	0
3	ROS-Kinetic	1	0	0
4	Gazebo 9.0.0	1	0	0
5	Sketchup Free	1	0	0
6	Honorarios	300	6,25€	1875€
TOTAL				**1987.35€**

Tabla 5 Presupuesto del proyecto

10 REFERENCIAS

[1] L. C. Davis, Effect of adaptive cruise control systems on traffic flow, 2004.

[2] «INTRODUCCIÓN DE ROS,» Erle Robotics, [En línea]. Available: http://erlerobotics.com/blog/ros-introduction-es/.

[3] «ROS/Concepts,» Institute for Systems and Robotics, [En línea]. Available: http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)Concepts.html.

[4] «Core Components,» ROS.org, [En línea]. Available: http://www.ros.org/core-components/#robot_specific_features.

[5] «Introducción a ROS,» de *Manual de ROS*.

[6] «Why Gazebo?,» Gazebo, [En línea]. Available: http://gazebosim.org/.

[7] «Extensible Markup Language,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Extensible_Markup_Language.

[8] «XML Robot Description Format (URDF),» ROS.org, [En línea]. Available: http://wiki.ros.org/urdf/XML/model.

[9] «URDF Reference,» enesbot.me, [En línea]. Available: http://enesbot.me/urdf-reference.html.

[10] «<Joint> element,» ROS.org, [En línea]. Available: http://wiki.ros.org/urdf/XML/joint.

[11] «URDF Transmissions,» ROS.org, [En línea]. Available: http://wiki.ros.org/urdf/XML/Transmission.

[12] "Tutorial: Using a URDF in Gazebo," gazebosim.org , [Online]. Available: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[13] «Launch Files,» Clear Path Robotics, [En línea]. Available: http://www.clearpathrobotics.com/assets/guides/ros/Launch%20Files.html.

[14] «Github,» ackermann_vehicle, [En línea]. Available: https://github.com/jbpassot/ackermann_vehicle.

[15] «Github,» rbcar_sim, [En línea]. Available: https://github.com/RobotnikAutomation/rbcar_sim.

[16] «Robotnik,» RB-CAR, [En línea]. Available: https://www.robotnik.es/robots-moviles/rbcar/.

[17] «velo2cam_gazebo,» Github, [En línea]. Available: https://github.com/beltransen/velo2cam_gazebo.

[18] «Gazebo plugins in ROS,» Gazebo.org, [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_gzplugins.

[19] «map server,» [En línea]. Available: http://wiki.ros.org/map_server.

[20] «Odometría,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Odometr%C3%ADa.

[21] «HP,» El corte inglés, [En línea]. Available: https://www.elcorteingles.es/electronica/A22557228-portatil-hp-15-bs027ns-i5-16-gb-1-tb-radeon-530-4-gb/.

[22] «XACRO,» ROS.org, [En línea]. Available: http://wiki.ros.org/xacro.